Internship report: Quotient RDF graph summarization

Pawel Guzewicz

Université Paris-Saclay
Télécom ParisTech

Master 2 Data and Knowledge

INTERNSHIP REPORT
QUOTIENT RDF GRAPH SUMMARIZATION

Paweł Guzewicz

Internship advisor: Ioana Manolescu

Scientific advisor: Benoît Groz

Palaiseau, September 3, 2018

Contents

Acknowledgements

I would like to express my deep gratitude to professor Ioana Manolescu, my research supervisor, for her patient guidance, sharing her experience and giving me plenty of advice. Through constructive suggestions and useful criticism she helped me develop new skills, not only scientific but also interpersonal. A number of recommendations and motivating comments greatly improved my knowledge and the quality of my work. Teaching me how to write a scientific paper, demystify the research workflow, and finally working together on projects with ambition and commitment, made me feel an important part of the team and a member of the research community. Last but not least, I want to thank her for encouraging me to pursue a PhD and to continue improving my French. Merci!

Chapter 1

Introduction

1.1 Organizational aspects

This report describes the main results obtained during my 6 month research internship at Inria Saclay. I worked under the supervision of Ioana Manolescu and the topic was "Exploring RDF graphs: expressivity and scale". Before the beginning of my employment I had already been working with my supervisor starting from November 2017. The internship was therefore a continuation of our scientific collaboration in a full-time setting.

The main focus of this research was the development of theoretical and practical solutions for summarization of heterogeneous graphs, RDF graphs in particular. Throughout my internship I have been involved in three projects: "Quotient RDF Summaries Based on Type Hierarchies", "Compact Summaries of Rich Heterogeneous Graphs" and "Distributed RDF Graph Summarization".

1.2 Scientific outline

My internship focuses on large data graphs with heterogeneous structure, possibly featuring typed data and an ontology, such as RDF graphs. The aim is to find a compact yet informative representation of the graphs. Before my arrival in the team in March 2018, two new graph node equivalence relations, that lead to quotient summaries, had been introduced in a technical report [9]. The authors also presented extensions which capture the semantic information encoded in the RDF graph that provide a special way to treat the types and the ontology, along with an RDF graph summarization framework. Most of my internship was devoted to novel summarization algorithms. Further, I also worked on an extended, novel treatment of the typed triples during the summarization.

More specifically, my internship work makes the following contributions.

- First, a set of novel algorithms for computing the summary of the graph in the centralized setting was devised jointly by my supervisor and myself. In particular, among those algorithms some are global, and some are incremental. I will refer to them later as *the centralized algorithms*. The implementation of those algorithms has been started from scratch in March 2018 as a Java/Maven project by my supervisor and I gradually took over the responsibility for it. We deployed the stable version (quotientSummary 1.6) on June 30, 2018. The detailed explanation of my contribution can be found in Chapter 4 and in the research report [4] which I co-authored. Large parts of this report borrow content from there, namely Chapters 2, 4, and major parts of Chapter 5. We devoted particular attention to the interplay between summarization and saturation of the graph. The saturation enriches the semantic information present in the graph thanks to the entailment rules applied on the input graph, assuming the ontology is present. One of the interesting theoretical results for the saturation, an efficient procedure called shortcut, was tested against the direct method handling the saturation in the process of summarization, when applicable. The details can be found in the experimental study (Chapter 5).

- Second, I have been working on a novel type-hierarchy-based quotient summarization method. It is an orthogonal yet closely related contribution that only me and my supervisor were in charge of. Strictly speaking, this work has started prior to the beginning of my internship as a collaboration with my current supervisor and led to the publication in DESWeb'2018 workshop associated with ICDE conference. I presented it on April 16 in Paris. The content of Chapter 3 is borrowed from this publication. I didn't have enough time to experiment with this approach during my internship.
- Third, I have worked to devise original, distributed approach for computing quotient summaries, using Spark framework. In collaboration with my supervisor, I have designed novel distributed algorithms, in particular aiming to address the practical big data applications of the RDF graph summaries. I have implemented those algorithms using Spark with Scala API, later on referred to as *the distributed algorithms*. An experimental study of their performance and possibly a paper submission based on them are part of our future work.

1.3 Report structure

This document is organized as follows. Chapter 2 contains preliminaries, summarizes the prior works and introduces the necessary notation. We recall the definitions of data graphs and the quotient summarization framework. In Chapter 3 we describe a new RDF graph summarization technique based on type hierarchies. Afterwards, Chapter 4 provides the detailed description of the algorithms for building quotient summaries of RDF graphs. Chapter 5 presents the experimental study of the performance of those algorithms, with their trade-offs explained. In Chapter 6 we comment on the related work. Finally, in Chapter 7 we conclude the results.

Chapter 2

Preliminaries

This chapter presents the background knowledge and material necessary to understand all the concepts presented in this report. Section 2.1 introduces the data graphs that are the subject of this research. Section 2.2 recalls the quotient summarization framework. The following three sections (2.3, 2.4, 2.5) provide detailed description of the process of summarization gradually adding new features enabling us to target the most general type of graphs, namely RDF graphs.

2.1 Data graphs

Our work is targeted to *directed graphs, with labeled nodes and edges*. This includes those described in RDF [39], the W3C standard for representing Web data. However, RDF graphs attach special interpretation to certain kinds of edges: (i) *type* edges may be used to attach type information to a data node; (ii) *ontology* edges may describe application-domain knowledge as relationships that hold between edge labels and/or node types.

Below, we introduce the useful terminology for RDF graphs since they are *the most general class of graphs for which our summarization methods apply*. We also isolate significant subsets of such graphs, which will be handled differently during summarization.

An *RDF graph* is a set of *triples* of the form `s p o`. A triple states that its *subject* `s` has the *property* `p`, and the value of that property is the *object* `o`. We consider only well-formed triples, as per the RDF specification [39], using uniform resource identifiers (URIs), typed or untyped literals (constants) and blank nodes (unknown URIs or literals). Blank nodes are essential features of RDF allowing to support *unknown URI/literal tokens*. These are conceptually similar to the labeled nulls or variables used in incomplete relational databases [1], as shown in [18].

Notations. We use `s`, `p`, and `o` as placeholders for subjects, properties and objects, respectively.

Figure 2.1 (top) shows how to use triples to describe resources, that is, to express class (unary relation) and property (binary relation) assertions. The RDF standard [39] has a set of *built-in classes and properties*, as part of the rdf: and rdfs: pre-defined namespaces. We use these namespaces exactly for these classes and properties, e.g., $\mathrm{rdf{:}type}$ specifies the class(es) to which a resource belongs.

As our running example, Figure 2.2 shows a sample RDF graph. Black and violet edges encode data and type respectively, e.g., node n_1 has property a whose object (value) is a_1, n_1 is of class (or has type) C_1, etc.

RDF Schema (RDFS). RDFS allows enhancing the descriptions in RDF graphs by declaring *ontological constraints* between the classes and the properties they use. Figure 2.1 (bottom) shows the four kinds of RDFS constraints, and how to express them through triples. Here, "domain" denotes the first, and "range" the second attribute of every property. In Figure 2.2, the blue edges connecting boxed nodes are RDFS constraints: they state that C_1 and C_2 are subclasses of C, and that the domain of d is C_2.

RDFS constraints are interpreted under the *open-world assumption (OWA)* [1], i.e., as deductive constraints. *RDF entailment* is the mechanism through which, based on a set of explicit triples and some

entailment rules, implicit RDF triples are derived. For instance, in Figure 2.2, node n_1 is of type C_1, which is a subclass of C. Through RDF entailment with the subclass constraint in Figure 2.1, we obtain the *implicit (entailed) triple* n_1 *type* C stating that n_1 is of class C. Similarly, n_2 and n_4 have property d whose domain is C_2 (thanks to the domain constraint in Figure 2.1), thus n_2 and n_4 are also of class C_2. Further, because C_2 is a subclass of C, they are also of class C.

In general, a triple $\mathtt{s\ p\ o}$ is entailed by a graph $\mathtt{G}$, denoted $\mathtt{G} \vdash_{\text{RDF}} \mathtt{s\ p\ o}$, if and only if there is a sequence of applications of entailment rules that leads from $\mathtt{G}$ to $\mathtt{s\ p\ o}$ (where at each step, triples previously entailed are also taken into account).

RDF graph saturation. Given a set of entailment rules, the *saturation* (a.k.a. closure) of an RDF graph $\mathtt{G}$, is defined as the fixpoint obtained by repeatedly adding to $\mathtt{G}$ the triples derived using the entailment rules; we denote it $\mathtt{G}^{\infty}$. For the four constraints shown in Figure 2.1, which we consider throughout this work, the saturation of $\mathtt{G}$, is finite, unique (up to blank node renaming), and does not contain implicit triples (they have all been made explicit by saturation). An obvious connection holds between the triples entailed by a graph $\mathtt{G}$ and its saturation: $\mathtt{G}$ entails (leads to, has as logical consequence) the triple $\mathtt{s\ p\ o}$ if and only if $\mathtt{s\ p\ o} \in \mathtt{G}^{\infty}$. It is important to note that the semantics of an RDF graph is its saturation [39]. In particular, when querying an RDF graph, the *answer* to the query should be computed both from its explicit and its implicit triples.

For presentation purposes, we may use a *triple-based* or a *graph-based* representation of an RDF graph:

1. Triple-based representation of an RDF graph. We see $\mathtt{G}$ as a *union of three edge-disjoint subgraphs* $\mathtt{G} = \langle \mathtt{D_G}, \mathtt{S_G}, \mathtt{T_G} \rangle$, where: (i) $\mathtt{S_G}$, the **schema** component, is the set of all $\mathtt{G}$ triples whose properties are *subclass*, *subproperty*, *domain* or *range*; we depict such triples with blue edges; (ii) $\mathtt{T_G}$, the **type** component, is the set of *type* triples from $\mathtt{G}$; we show them in violet; (iii) $\mathtt{D_G}$, the **data** component, holds all the remaining triples of $\mathtt{G}$; we display them in black. Note that each union of the $\mathtt{D_G}$, $\mathtt{S_G}$, and $\mathtt{T_G}$ components is an RDF graph by itself.

Further, we call *data property* any property $\mathtt{p}$ occurring in $\mathtt{D_G}$, and *data triple* any triple in $\mathtt{D_G}$.

2. The graph-based representation of an RDF graph. As per the RDF specification [39], *the set of nodes* of an RDF graph is the set of subjects and objects of triples in the graph, while its *edges* correspond to its triples. We define three categories of RDF graph nodes: (i) a *class node* is any node whose URI appears as subject or object of a *subclass* triple, or object of a *domain* or *range* or *type* triple; we show them in blue boxes; (ii) a *property node* is any node whose URI appears as subject or object of a *subproperty* triple, or subject of a *domain* or *range* triple, or property of a triple[1]; in Figure 2.2, d is a property node. We also show them in blue boxes; (iii) a *data node* is any node that is neither a class nor a property node. We show them in black. Note that the sets of class nodes and of property nodes may intersect (indeed, nothing in the RDF specification forbids it). However, data nodes are disjoint from both class and property nodes.

[1] A property node must be a *node*, i.e., merely appearing in a property position does not make an URI a property node; for this, the URI needs to appear *as a subject or object* in a triple of the graph.

Assertion	Triple	Relational notation
Class	$\mathtt{s}$ rdf:type $\mathtt{o}$	$\mathtt{o(s)}$
Property	$\mathtt{s\ p\ o}$	$\mathtt{p(s, o)}$

Constraint	Triple	OWA interpretation
Subclass	$\mathtt{s}$ *subclass* $\mathtt{o}$	$\mathtt{s} \subseteq \mathtt{o}$
Subproperty	$\mathtt{s}$ *subproperty* $\mathtt{o}$	$\mathtt{s} \subseteq \mathtt{o}$
Domain typing	$\mathtt{p}$ *domain* $\mathtt{o}$	$\Pi_{\text{domain}}(\mathtt{p}) \subseteq \mathtt{o}$
Range typing	$\mathtt{p}$ *range* $\mathtt{o}$	$\Pi_{\text{range}}(\mathtt{p}) \subseteq \mathtt{o}$

Figure 2.1: RDF (top) & RDFS (bottom) statements.

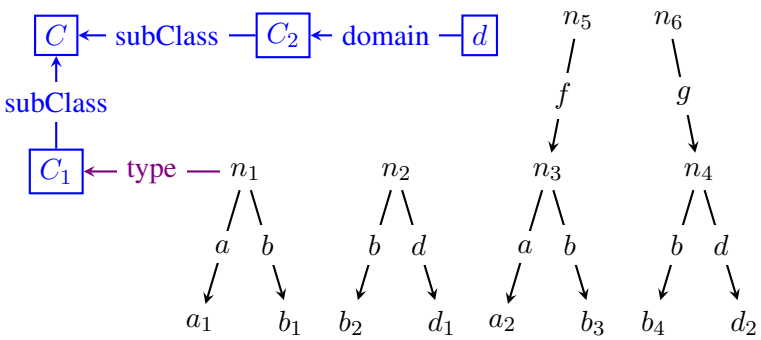


Figure 2.2: Sample RDF graph $\mathtt{G} = \langle \mathtt{D_G}, \mathtt{S_G}, \mathtt{T_G} \rangle$: $\mathtt{D_G}$ edges are shown in black, $\mathtt{S_G}$ edges in blue, $\mathtt{T_G}$ edges in violet.

Size and cardinality notations. We denote by $|\mathtt{G}|_n$ the number of nodes in a graph $\mathtt{G}$, and by $|\mathtt{G}|$ its number of edges. Further, for a given attribute $x \in \{\mathtt{s}, \mathtt{p}, \mathtt{o}\}$ and graph $\mathtt{G}$, we note $|\mathtt{G}|^0_x$ the number of distinct values of the attribute x within $\mathtt{G}$. For instance, $|\mathtt{D_G}|^0_\mathtt{p}$ is the number of distinct properties in the data component of $\mathtt{G}$.

We will rely on the graph, respectively, the triple-based representation when each is most natural for the presentation; accordingly, we may use *triple* or *edge* interchangeably to denote a graph edge.

2.2 Summarization framework

We recall the classical notion of *graph quotient*, on which many graph summaries, including ours, are based. In particular, we recall quotients based on *bisimilarity*, and show that their very nature makes them ill-suited to summarize heterogeneous graphs.

Graph quotients. Given a data graph $\mathtt{G}$ and an equivalence relation[2] $\equiv$ over the node of $\mathtt{G}$, the quotient of $\mathtt{G}$ by $\equiv$, denoted $\mathtt{G}_{/\equiv}$, is the graph having (i) a node for each equivalence class of $\equiv$ (thus, for each set of equivalent $\mathtt{G}$ nodes); and (ii) for each edge $n_1 \xrightarrow{a} n_2$ in $\mathtt{G}$, an edge $m_1 \xrightarrow{a} m_2$, where m_1, m_2 are the quotient nodes corresponding to the equivalence classes of n_1, n_2 respectively.

Many known graph summaries, e.g., [32, 25, 12, 37, 28, 14, 17] are quotient-based; they differ in their equivalence relations $\equiv$. Quotient summaries have several desirable properties:

Size guarantees By definition, $\mathtt{G}_{/\equiv}$ is guaranteed to have at most as many nodes and edges as $\mathtt{G}$. Some non-quotient summaries, e.g., Dataguides [19], cannot guarantee this.

Property completeness denotes the fact that every property (edge label) from $\mathtt{G}$ is present on summary edges. This is helpful to users approaching a graph dataset for the first time[3]; it is for this scenario precisely that our summaries are used in LODAtlas [34].

Structural representativeness is the following property: for any query q that has answers on $\mathtt{G}$, its *structure-only* version q', which copies all the graph patterns of q but erases its possible selections on nodes as well as counting predicates, is guaranteed to have answers on $\mathtt{G}_{/\equiv}$.

For instance, if q_1 is "find all nodes that are target of an f edge and source of a b and a d edge" on the graph in Figure 2.2, then q_1' is the same as q_1. If the query q_2 is "find all nodes whose labels contain "Alice", having exactly one (not more) incoming f edge and exactly one outgoing b edge", the query q_2' asks for "all nodes having incoming f and outgoing b edges". Thanks to representativeness, quotient

[2] An equivalence relation $\equiv$ is a binary relation that is reflexive, i.e., $x \equiv x$, symetric, i.e., $x \equiv y \Rightarrow y \equiv x$, and transitive, i.e., $x \equiv y$ and $y \equiv z$ implies $x \equiv z$ for any x, y, z.

[3] Most complete summarization methods, including ours, can be adapted to reflect e.g., only properties above a certain frequency threshold in the graph etc. We will not consider this further.

summaries can be used to prune empty-answer queries: if $q'(\mathtt{G}_{/\equiv})$ has no answers, then q has no answers on $\mathtt{G}$. Since the summary is often much smaller than $\mathtt{G}$, pruning is very fast and saves useless query evaluation effort on $\mathtt{G}$.

To enjoy the above advantages, in this work, *a summary of* $\mathtt{G}$ *is its quotient through some equivalence relation*.

Two extreme quotient summaries help to see the trade-offs in this context. First, let $\top$ denote the equivalence relation for which all nodes are equivalent: then, $\mathtt{G}_{/\top}$ has a single node with a loop edge to itself for each distinct property in $\mathtt{G}$. This summary collapses (and loses) most of the graph structure. Now, let $\bot$ denote the equivalence relation for which each node is only equivalent to itself. Then, $\mathtt{G}_{/\bot}$ is isomorphic to $\mathtt{G}$ for any graph $\mathtt{G}$; it preserves all the structure but achieves no summarization.

Bisimulation-based summaries. Many known structural quotient summaries, e.g., [32, 12, 26, 15] are based on bisimilarity [23]. Two nodes n_1, n_2 are *forward and/or backward bisimilar* (denoted $\equiv_{\mathtt{fw}}$, $\equiv_{\mathtt{bw}}$ and $\equiv_{\mathtt{fb}}$) iff for every $\mathtt{G}$ edge $n_1 \xrightarrow{a} m_1$, $\mathtt{G}$ also comprises an edge $n_2 \xrightarrow{a} m_2$, such that m_1 and m_2 are also forward and/or backward bisimilar, respectively. The $\equiv_{\mathtt{fw}}$ and $\equiv_{\mathtt{bw}}$ relations only take into account the paths outgoing from (resp. only the paths incoming to) the nodes. The **symmetry** of $\mathtt{G}_{/\mathtt{fb}}$ is an advantage, as it makes it more resistent to minor modeling differences in the data. For instance, let t' be the triple a *hasAuthored* p and t'' be p *hasAuthor* a, which essentially represent the same information. Triple t' would impact the nodes to which a is $\equiv_{/\mathtt{fw}}$, while it would not impact the nodes to which a is $\equiv_{/\mathtt{bw}}$; symetrically, t'' would impact the $\equiv_{/\mathtt{bw}}$ class of p but not its $\equiv_{/\mathtt{fw}}$ class. In contrast, $\equiv_{/\mathtt{fb}}$ reflects this information whether it is modeled in one direction or in the other.

We denote the bisimulation based summaries $\mathtt{G}_{/\mathtt{fw}}$ (forward), $\mathtt{G}_{/\mathtt{bw}}$ (backward) and $\mathtt{G}_{/\mathtt{fb}}$ (forward and backward), respectively. They tend to be **large** because *bisimilarity is rare in heterogeneous data graphs*. For instance, each node of the graph in Figure 2.2 is only $\equiv_{/\mathtt{fb}}$ to itself, thus $\equiv_{/\mathtt{fb}}$ is useless for summarizing it; our experiments in Section 5.1 confirm this on many graphs. To mediate this problem, k**-bisimilarity** has been introduced [27], whereas nodes are k-forward (and/or backward) bisimilar iff their adjacent paths *of length at most* k are identical; the smaller k is, the more permisive the equivalence relation.

One drawback of k-bisimilarity is that it requires users to guess the k value leading to the best compromise between compactness (which favors low k; note that $k = 0$ leads exactly to $\mathtt{G}_{/\top}$) and structural information in the summary (high k). Further, *even* 1*-bisimilarity is hard to achieve* in heterogeneous graphs. For instance, Figure 2.3 shows the $\mathtt{1fb}$ summary of the sample graph in Figure 2.2[4]. Nodes in the bottom row of $\mathtt{G}$, which only have incoming a, b, respectively d edges, and have no outgoing edges, are summarized together. However, none of n_1, n_2, n_3 and n_4 are equivalent, because of the presence/absence of a and d edges, and of their possible incoming edges.

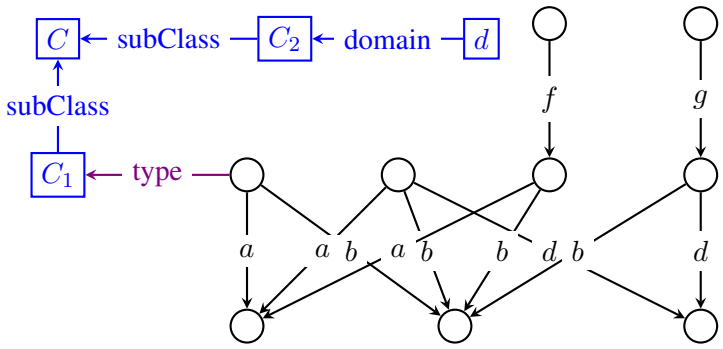


Figure 2.3: $\mathtt{1fb}$ summary of the RDF graph in Figure 2.2.

Any structural equivalence relation cuts a trade-off between compactness and structure preservation. Below, we introduce two relations leading to compact summaries that *in addition* cope well with the

[4]In Figure 2.3, type and schema triples are summarized according to the method we propose below. However, the treatment of these triples is orthogonal to the interesting aspects of this example.

data heterogeneity frequently encountered in RDF (and other) graphs.

Other interesting properties of such relations are: the complexity of building the corresponding summary, and of updating it to reflect graph updates. Further, the presence of type and schema (ontology) triples has not been formally studied in quotient graph summarization.

2.3 Data graph summarization

We first consider graphs made of *data triples* only, thus of the form $\mathtt{G} = \langle \mathtt{D}_\mathtt{G}, \varnothing, \varnothing \rangle$. We define the novel notion of *property cliques* in Section 2.3.1; building on them, we devise new graph node equivalence relations and corresponding graph summaries in Section 2.3.2. Summarization will be generalized to handle also *type triples* in Section 2.4, and *type and schema triples* in Section 2.5.

2.3.1 Data property cliques

We are interested in defining equivalence relations between two data nodes, that can cope with the heterogeneity present in many real-life data graphs. For example, up to 14 data properties (such as title, author, year, but also note, etc.) are used to describe conference papers in a graph version of the DBLP bibliographic database. Each paper has a certain subset of these 14 properties, and has some of them, e.g., authors, with multiple values; we counted more than 130 such distinct property subsets in a small (8MB) fragment of DBLP. To avoid the "noise" introduced by such structural heterogeneity, we need node equivalence relations that look *beyond* it, and consider that all the nodes corresponding to conference publications are equivalent.

To do that, we first focus on the way data properties (edge labels) are organized in the graph. The simplest relation that may exist between two properties is *co-occurrence*, when a node is the source (or target) of two edges carrying the two labels. However, in heterogeneous RDF graphs such as DBLP, two properties, say author and title, may co-occur on a node n, while another node n' has title, year, and howpublished: we may consider *all* these properties (author, title, year and howpublished) related, as they (directly or *transitively*) co-occur on some nodes. Formally:

Definition 1. (PROPERTY RELATIONS AND CLIQUES) *Let* $\mathtt{p}_1$, $\mathtt{p}_2$ *be two data properties in* $\mathtt{G}$*:*

1. $\mathtt{p}_1, \mathtt{p}_2 \in \mathtt{G}$ *are* source-related *iff either: (i) a data node in* $\mathtt{G}$ *is the subject of both* $\mathtt{p}_1$ *and* $\mathtt{p}_2$*, or (ii)* $\mathtt{G}$ *holds a data node that is the subject of* $\mathtt{p}_1$ *and a data property* $\mathtt{p}_3$*, with* $\mathtt{p}_3$ *and* $\mathtt{p}_2$ *being source-related.*

2. $\mathtt{p}_1, \mathtt{p}_2 \in \mathtt{G}$ *are* target-related *iff either: (i) a data node in* $\mathtt{G}$ *is the object of both* $\mathtt{p}_1$ *and* $\mathtt{p}_2$*, or (ii)* $\mathtt{G}$ *holds a data node that is the object of* $\mathtt{p}_1$ *and a data property* $\mathtt{p}_3$*, with* $\mathtt{p}_3$ *and* $\mathtt{p}_2$ *being target-related.*

A maximal set of data properties in $\mathtt{G}$ *which are pairwise source-related (respectively, target-related) is called a* source *(respectively,* target*)* property clique*.*

In the graph in Figure 2.2, properties a and b are source-related due to n_1 (condition 1. in the definition). Similarly, b and d are source-related due to n_2; consequently, a and d are source-related (condition 2.). Thus, a source clique of this graph is $SC_1 = \{a, b, d\}$. Table 2.1 shows the target and source cliques of all data nodes from Figure 2.2.

It is easy to see that the set of non-empty source (or target) property cliques is a *partition over the data properties of* $\mathtt{G}$. Further, if a node $n \in \mathtt{G}$ is source of some data properties, they are all in the same source clique; similarly, all the properties of which n is a target are in the same target clique. This allows us to refer to *the source (or target) clique of* n, denoted $SC(n)$ and $TC(n)$.

n	n_1	n_2	n_3	n_4	n_5	n_6
$SC(n)$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{f\}$	$\{g\}$
$TC(n)$	$\varnothing$	$\varnothing$	$\{f\}$	$\{g\}$	$\varnothing$	$\varnothing$

n	a_1	a_2	b_1	b_2	b_3	d_1	d_2
$SC(n)$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
$TC(n)$	$\{a\}$	$\{a\}$	$\{b\}$	$\{b\}$	$\{b\}$	$\{d\}$	$\{d\}$

Table 2.1: Source and target cliques of G nodes (Figure 2.2).

2.3.2 Strong and weak node equivalences

Building on property cliques, we define two main *node equivalence relations* among the data nodes of a graph G:

Definition 2. (STRONG EQUIVALENCE) *Two data nodes of* G *are* strongly equivalent*, denoted* $n_1 \equiv_S n_2$*, iff they have the same source* and *target cliques.*

Strongly equivalent nodes have the same structure of *incoming and outgoing* edges. In Figure 2.2, nodes n_1, n_2 are *strongly* equivalent to each other, and so are e.g., a_1, a_2, b_1, b_2 and b_3 etc.

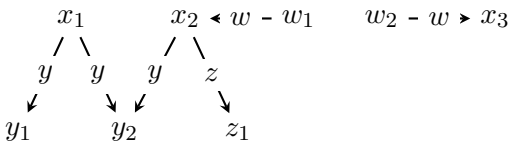


Figure 2.4: Sample weakly equivalent nodes: x_1, x_2, x_3.

A second, weaker notion of node equivalence could request only that equivalent nodes share the same *incoming or outgoing* structure, i.e., they share the same source clique or the same target clique. Figure 2.4 illustrates this. Nodes x_1, x_2 have the same source clique because they both have outgoing y edges. Further, x_2 and x_3 have the same target clique because both have incoming w edges. Since equivalence must be transitive, it follows that x_1 and x_3 must also be considered weakly equivalent, since they "follow the same pattern" of having at least one incoming w edge, or at least one outgoing y or z edge, and no other kinds of edges. Formally:

Definition 3. (WEAK EQUIVALENCE) *Two data nodes are* weakly equivalent*, denoted* $n_1 \equiv_W n_2$*, iff:* (i) *they have the same non-empty source or non-empty target clique, or* (ii) *they both have empty source and empty target cliques, or* (iii) *they are both weakly equivalent to another node of* G*.*

It is easy to see that $\equiv_W$ and $\equiv_S$ are equivalence relations and that strong equivalence implies weak equivalence.

In Figure 2.2, $n_1, \ldots, n_4$ are *weakly* equivalent to each other due to their common source clique SC_1; a_1, a_2 are weakly equivalent due to their common target clique etc.

2.3.3 Weak and strong summarization

Notation: representation function. We say the summary node of $G_{/\equiv}$ corresponding to the equivalence class of a G node n represents n, and denote it $f_{\equiv}(n)$ or simply $f(n)$ when this does not cause confusion. We call $f_{\equiv}$ the *representation function* of the equivalence relation $\equiv$ over G.

Weak summarization. The first summary we define is based on weak equivalence:

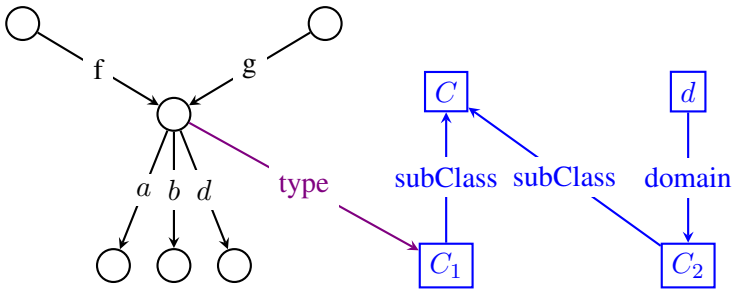


Figure 2.5: Weak summary of the RDF graph in Figure 2.2.

Definition 4. (WEAK SUMMARY) *The* weak summary *of a data graph* $\mathtt{G}$*, denoted* $\mathtt{G}_{/\mathtt{W}}$*, is its quotient graph w.r.t. the weak equivalence relation* $\equiv_{\mathtt{W}}$.

The the graph in Figure 2.2 is depicted by the black nodes and edges in Figure 2.5[5]; summary nodes are shown as unlabeled circles, to denote that they are anonymous (new) nodes, each of which represents one or more $\mathtt{G}$ nodes. The central one represents n_1, n_2, n_3 and n_4. Its outgoing edges go towards nodes representing, respectively, a_1 and a_2; b_1, b_2 and b_3; finally, d_1 and d_2. Its incoming edges come from the representative of n_5 (which was a source of an f edge in $\mathtt{G}$), respectively from the representative of n_6 (source of g).

The weak summary has the following important property:

Proposition 1. (UNIQUE DATA PROPERTIES) *Each* $\mathtt{G}$ *data property appears exactly once in* $\mathtt{G}_{/\mathtt{W}}$.

Proof. First, note that any two weak summary nodes n_1, n_2 cannot be targets of the same data property. Indeed, if such a data property $\mathtt{p}$ existed, let TC be the target clique it belongs to. By the definition of the weak summary, n_1 corresponds to a set of (disjoint) target cliques STC_1, which includes TC, and a set of disjoint source cliques SSC_1. Similarly, n_2 corresponds to a set of (disjoint) target cliques STC_2, which includes TC, and a set of disjoint source cliques SSC_2. The presence of TC in STC_1 and STC_2 contradicts the fact that different equivalence classes of $\mathtt{G}$ nodes correspond to disjoint sets of target cliques. The same holds for the sets of properties of which weak summary nodes are sources. Thus, any data property has at most one source and at most one target in $\mathtt{G}_{/\mathtt{W}}$. Further, by the definition of the summary as a quotient, every data property present in $\mathtt{G}$ also appears in the summary. Thus, there is exactly one $\mathtt{p}$-labeled edge in $\mathtt{G}_{/\mathtt{W}}$ for every data property in $\mathtt{G}$. □

Importantly, the above Proposition 1 warrants that $|\mathtt{G}_{/\mathtt{W}}|$, the number of edges in $\mathtt{G}_{/\mathtt{W}}$, is exactly the number of distinct data properties in $\mathtt{G}$. This observation is used in our weak summarization algorithms (Section 4.1). By definition of a quotient summary (Section 2.2), *this is the smallest number of edges a summary may have* (since it has at least one edge per each distinct property in $\mathtt{G}$). Thus, $\mathtt{G}_{/\mathtt{W}}$ is a minimal-size quotient summary (like $\mathtt{G}_{/\mathtt{T}}$ from Section 2.2, but much more informative than it). As our experiments show, $|\mathtt{G}_{/\mathtt{W}}|$ is typically 3 to 6 orders of magnitude smaller than $|\mathtt{G}|$.

Strong summarization. Next, we introduce:

Definition 5. (STRONG SUMMARY) *The* strong summary *of the graph* $\mathtt{G}$*, denoted* $\mathtt{G}_{/\mathtt{S}}$*, is its quotient graph w.r.t. the strong equivalence relation* $\equiv_{\mathtt{S}}$.

The strong summary of the graph of Figure 2.2 is shown by the black edges and nodes in Figure 2.6. Similarly to the weak summary (Figure 2.5), the strong one features a single node source of a, respectively, b, d, f and g edges. However, differently from $\mathtt{G}_{/\mathtt{W}}$, the strong summary splits the data nodes whose source clique is $\{a, b, d\}$ in *three equivalence classes:* n_1 and n_2 have the empty target clique, while that of n_2 is $\{f\}$ and that of n_3 is $\{g\}$. Thus, two data nodes represented by the same

[5]The violet and blue edges serve our discussion later on.

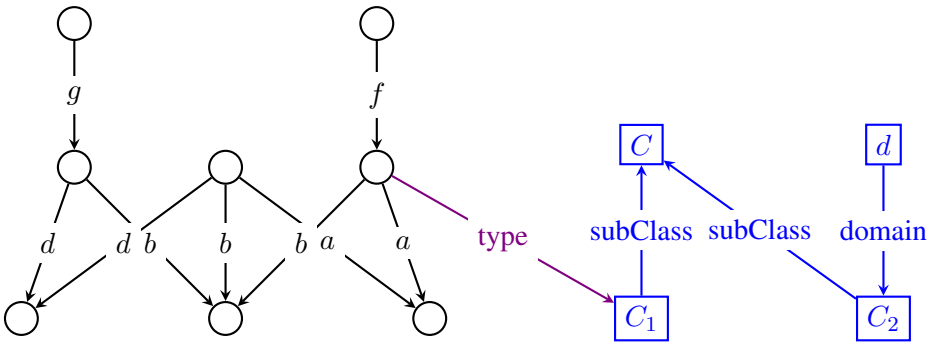


Figure 2.6: Strong summary of the RDF graph in Figure 2.2.

strong summary node have similar structure both in their inputs and outputs; in contrast, a weak summary (recall Figure 2.4) represents together nodes having similar structure in their inputs *or* outputs, *or* which are both equivalent to another common node. As we can see, *strong summarization leads to finer-granularity* summaries. An effect of this finer granularity is that in $\mathtt{G}_{/\mathtt{S}}$, several edges may have the same label, e.g., there are three edges labeled b in Figure 2.5 (whereas for $\mathtt{G}_{/\mathtt{W}}$, as stated in Proposition 1, this is not possible). Our experiments (Section 5.1) show that while $\mathtt{G}_{/\mathtt{S}}$ is often somehow larger than $\mathtt{G}_{/\mathtt{W}}$, it still remains many orders of magnitude smaller than the original graph.

By definition of $\equiv_{\mathtt{S}}$, equivalent nodes have the same source clique and the same target clique. This leads to:

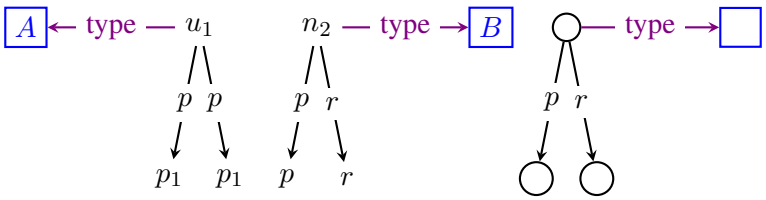
Proposition 2. (STRONG SUMMARY NODES AND G CLIQUES) $\mathtt{G}_{/\mathtt{S}}$ *has exactly one node for each source clique and target clique of a same node* $n \in \mathtt{D}_{\mathtt{G}}$.

Proposition 2 is exploited by the implementations of our strong summarization algorithms (Section 4.1).

2.4 Typed data graph summarization

We generalize our approach to summarize graphs with data and type triples, thus of the form $\mathtt{G} = \langle \mathtt{D}_{\mathtt{G}}, \mathtt{T}_{\mathtt{G}}, \varnothing \rangle$.

Starting from an equivalence relation $\equiv$ defined over *data* nodes, in order to summarize $\mathtt{D}_{\mathtt{G}} \cup \mathtt{T}_{\mathtt{G}}$, two questions must be answered: (i) how should $\equiv$ be extended on class nodes (such as C_1 in Figure 2.2)? and (ii) how should the type edge(s) of a node n be taken into account when determining to whom n is equivalent? Below, we answer these questions for *any* equivalence relation $\equiv$, then instantiate our answer to the weak and strong relations we defined.



To study the first question, consider the sample graph above, and a possible summary of this graph at its right. Assume that the types A and B are considered equivalent. Quotient summarization represents them both by the summary node at the top right, which (like all summary nodes) is a "new" node, i.e., it is neither A nor B. Observe that this summary *compromises representativeness* for queries over both the data and the type triples: for instance, the query asking for "nodes of type A having property r" is empty on the summary (as type A has been conflated with type B) while it is non empty on the graph.

To avoid this, we argue that when moving from data to typed data graphs, any equivalence relation $\equiv$ between data nodes should be extended to class nodes as follows: 1. *any class node is only equivalent to itself* and 2. *any class node is only represented by itself*, hence a graph has the same class nodes as its summary.

We now consider the second question: how should $\equiv$ be extended to exploit not only the data but also the type triples? Note that two nodes may have similar incoming/outgoing data edges but different type edges, or vice-versa, the same types but very different data edges. We introduce two main alternatives below, then decline them for weak and strong summarization.

2.4.1 Data-then-type summarization

This approach consists of using an equivalence $\equiv$ defined based on data properties in order to determine which data nodes are equivalent and thus to build summary nodes, and data edges between them. Afterward, for each triple n *type* C in $\mathtt{G}$, add to $\mathtt{G}_{\equiv}$ a triple $f_{\equiv}(n)$ *type* C, where we recall that $f_{\equiv}(n)$ is the representative of n in $\mathtt{G}_{\equiv}$. This approach is interesting, e.g., when only some of the nodes have types (often the case in RDF graphs). In such cases, it makes sense to first group nodes according to their data edges, while still preserving the (partial) type information they have. *We extend the* $\mathtt{W}$, *respectively* $\mathtt{S}$ *summaries to type triples, by stating that they* (i) *represent each class node by itself; and* (ii) *follow a data-then-type approach*, as described above.

In Figure 2.5, the black and violet edges (including the C_1 node) depict the weak summary of the black and violet graph triples Figure 2.2. The type edge reads as: *at least one* of the nodes represented by its source, was declared of type C_1 in the input graph. Similarly, the black and violet edges in Figure 2.6 show the strong summary of the same subset of our sample graph.

To recap, in data-then-type summarization using $\equiv$, two data nodes are represented together iff they are $\equiv$ based on their incoming and outgoing data edges, while a class node is only equivalent to itself (and always represented by itself).

One more point needs to be settled. Some $\mathtt{T}_{\mathtt{G}}$ nodes may have types, but no incoming or outgoing data properties. Strong summarization represents all such nodes together, based on their $(\varnothing, \varnothing)$ pair of source and target cliques. For completeness, we extend weak summaries to also represent such nodes together, by a single special node denoted $N_{\varnothing}$.

2.4.2 Type-then-data summarization

This approach takes the opposite view that node types are more important when deciding whether nodes are equivalent. Observe that our framework (just like RDF) does not prevent a node from having *several* types. At the same time, representing a node by *each* of its types separately would violate the quotient summarization framework, because a quotient, by definition, represents each node exactly once. Thus, in type-then-data summarization, we extend a given equivalence relation $\equiv$ (based on data properties alone) as follows.

Definition 6. (TYPED EQUIVALENCE) Typed equivalence*, denoted* $\equiv_{\mathtt{T}}$*, is an equivalence relation over* $\mathtt{D}_{\mathtt{G}} \cup \mathtt{T}_{\mathtt{G}}$ *defined as follows: two data nodes* n_1 *and* n_2 *are type-equivalent, noted* $n_1 \equiv_{\mathtt{T}} n_2$*, iff they have exactly the same set of types in* $\mathtt{G}$*, which is non-empty; any class node is only equivalent to itself.*

Intuitively, typed equivalence performs a first-cut data node classification, *according to their sets of types*. In particular, all untyped nodes are equivalent to themselves. This enables the definition of type-then-data summaries as *double quotients*: first, quotient $\mathtt{G}$ by $\equiv_{\mathtt{T}}$; then, quotient the resulting graph by some data node equivalence *only on untyped nodes* (each left alone in an equivalence class of $\equiv_{\mathtt{T}}$), to group them according to their data edges.

Applied to weak summarization, this approach leads to:

Definition 7. (TYPED WEAK SUMMARY) *Let* $\equiv_{\mathtt{UW}}$ *(*untyped weak equivalence*) be an equivalence relation that holds between two data nodes* n_1, n_2 *iff* (i) n_1, n_2 *have no types in* $\mathtt{G}$ *and* (ii) $n_1 \equiv_{\mathtt{W}} n_2$*. The* typed

weak summary $\mathtt{G}_{/\mathtt{TW}}$ *of a graph* $\mathtt{G}$ *is the untyped-weak summary of the type-based summary of* $\mathtt{G}$*, namely* $(\mathtt{G}_{/\mathtt{T}})_{/\mathtt{UW}}$.

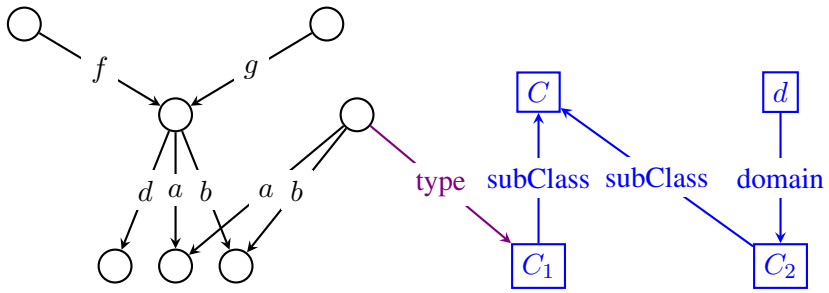


Figure 2.7: Typed weak summary of the graph in Figure 2.2.

In Figure 2.7, the black and violet edges depict the typed weak summary of the data and type edges of the sample graph in Figure 2.2. Unlike $\mathtt{G}_{/\mathtt{W}}$ (Figure 2.5), $\mathtt{G}_{/\mathtt{TW}}$ represents the node of type C_1 separately from the untyped ones having similar properties. This reflects the primordial role of types in type-then-data summarization.

In a similar manner, we define:

Definition 8. (TYPED STRONG SUMMARY) *Let* $\equiv_{\mathtt{US}}$ (untyped strong equivalence*) be an equivalence relation that holds between two data nodes* n_1, n_2 *iff (i)* n_1, n_2 *have no types in* $\mathtt{G}$ *and (ii)* $n_1 \equiv_{\mathtt{S}} n_2$*. The* typed strong summary $\mathtt{G}_{/\mathtt{TS}}$ *of an RDF graph* $\mathtt{G}$ *is defined as:* $(\mathtt{G}_{/\mathtt{T}})_{/\mathtt{US}}$.

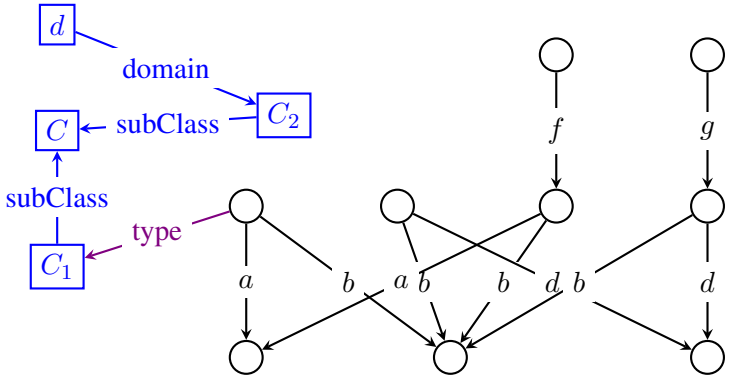


Figure 2.8: Typed strong summary of the graph in Figure 2.2.

In Figure 2.8, the black and violet edges depict the typed strong summary of the data and type edges of the sample graph in Figure 2.2. Unlike $\mathtt{G}_{/\mathtt{S}}$ (Figure 2.6), $\mathtt{G}_{/\mathtt{TS}}$ represents the node n_1 of type C_1 separately from n_2, which has no types.

2.5 RDF graph summarization

We consider now the summarization of general RDF graphs which may also have schema triples, i.e., of the form $\mathtt{G} = \langle \mathtt{D}_{\mathtt{G}}, \mathtt{T}_{\mathtt{G}}, \mathtt{S}_{\mathtt{G}} \rangle$.

2.5.1 Extending summarization

First, how to extend an equivalence relation $\equiv$ defined on data nodes (and extended as we discussed in Section 2.4 to class nodes, each of which is only equivalent to itself), to also cover *property nodes*, such as the boxed d node in Figure 2.2? Such property nodes provide important schema (ontology)

information, which describes how the properties and types present in the data relate to each other, and leads to implicit triples (recall Section 2.1). For the summary to preserve the semantics of the input graph, by an argument similar to the one at the beginning of Section 2.4, we impose that $\equiv$ be extended so that *any property node should (also) be equivalent only to itself*, and propose that in any summary, *any property node should be represented by itself.* As a consequence, since any class or schema node is equivalent only to itself and represented only by itself:

Proposition 3. (SCHEMA PRESERVATION THROUGH SUMMARIZATION) *For any equivalent relation $\equiv$ defined on data nodes and extended as specified above to class and property nodes, and any graph $\mathtt{G} = \langle \mathtt{D_G}, \mathtt{T_G}, \mathtt{S_G} \rangle$, it holds that: $\mathtt{S}_{\mathtt{G}_{\equiv}} = \mathtt{S_G}$, that is: the summary of $\mathtt{G}$ through $\equiv$ has exactly the schema triples of $\mathtt{G}$.*

This decision allows us to simply copy schema triples from the input graph to each of its summaries. Figures 2.5, 2.6, 2.7 and 2.8, considered in their entirety, show respectively full $\mathtt{G}_{/\mathtt{W}}$, $\mathtt{G}_{/\mathtt{S}}$, $\mathtt{G}_{/\mathtt{TW}}$ and $\mathtt{G}_{/\mathtt{TS}}$ summaries of the sample RDF graph in Figure 2.2.

2.5.2 Summarization versus saturation

As we explained in Section 2.1, the semantics of a graph $\mathtt{G}$ includes its explicit triples, but also its *implicit* triples which are not in $\mathtt{G}$, but hold in $\mathtt{G}^{\infty}$ due to ontological triples (such as the triples n_2 *type* C_2 and n_2 *type* C discussed in Section 2.1).

A first interesting question, then, is: how does saturation impact the summary of a graph? As Figure 2.1 shows, saturation adds data and type triples. Other entailment rules (see [18]) also generate schema triples, e.g., if C' is a subclass of C'' and C'' is a subclass of C''', then C' is also a subclass of C''' etc. Due to these extra edges, in general, $(\mathtt{G}^{\infty})_{/\equiv}$ and $\mathtt{G}_{/\equiv}$ are different. On one hand, their nodes may be different, but this is not the most interesting aspect, as summary nodes are just "representatives", i.e., they are labeled in a somehow arbitrary fashion. On the other hand (and this is much more meaningful), their graph structure may be different, as it results from graphs with different edges. To separate the mere difference of node IDs from the meaningful difference of graph structure, we define:

Definition 9. (STRONG ISOMORPHISM $\widehat{=}$) *A* strong isomorphism between two RDF graphs $\mathtt{G}_1, \mathtt{G}_2$, *noted $\mathtt{G}_1 \mathrel{\widehat{=}} \mathtt{G}_2$, is a graph isomorphism which is the identity for the class and property nodes.*

Intuitively, strongly isomorphic graphs (in particular, summaries) represent exactly the same information, while the identifiers of their non-class, non-property nodes (shown as unlabeled circles in our examples) may differ.

Next, one could wonder whether saturation *commutes with* summarization, that is, does $(\mathtt{G}^{\infty})_{/\equiv} \mathrel{\widehat{=}} (\mathtt{G}_{/\equiv})^{\infty}$ hold? If this was the case, it would lead to a likely more efficient method for computing the summary of $\mathtt{G}$'s full semantics, that is $(\mathtt{G}^{\infty})_{/\equiv}$, without saturating $\mathtt{G}$ (thus, without materializing all its implicit triples); instead, we would summarize $\mathtt{G}$ and then saturate the resulting (usually much smaller) graph. Unfortunately, Figure 2.9 shows that this is not always the case. For a given graph $\mathtt{G}$, it traces (top row) its weak summary $\mathtt{G}_{/\mathtt{W}}$ and its saturation $(\mathtt{G}_{/\mathtt{W}})^{\infty}$, whereas the bottom row shows $\mathtt{G}^{\infty}$ and its summary $(\mathtt{G}^{\infty})_{/\mathtt{W}}$. Here, saturation leads to b edges outgoing both r_1 and r_2 which makes them equivalent. In contrast, summarization *before* saturation represents them separately; saturating the summary cannot revert this decision, to unify them as in $(\mathtt{G}^{\infty})_{/\mathtt{W}}$ (recall from Section 2.1 that saturation can only add edges between $\mathtt{G}$ nodes).

[8] had introduced we the following three-step transformation aiming at obtaining $(\mathtt{G}^{\infty})_{/\equiv}$: first summarize $\mathtt{G}$; then saturate its summary; finally, summarize it again in order to build $((\mathtt{G}_{/\equiv})^{\infty})_{/\equiv}$. When this leads to $(\mathtt{G}^{\infty})_{/\equiv}$, the *shortcut* is said to hold:

Definition 10. (SHORTCUT) *We say* the shortcut holds *for a given RDF node equivalence relation $\equiv$ iff for any RDF graph $\mathtt{G}$, $(\mathtt{G}^{\infty})_{/\equiv}$ and $((\mathtt{G}_{/\equiv})^{\infty})_{/\equiv}$ are strongly isomorphic.*

In [9], the authors establish:

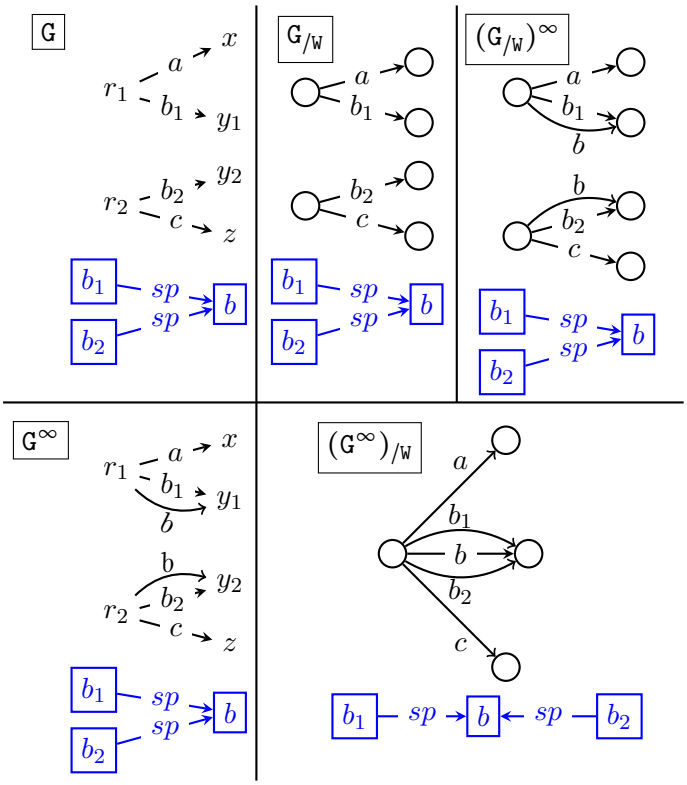


Figure 2.9: Saturation and summarization on a sample graph.

Theorem 1 (W shortcut)**.** *The shortcut holds for* $\equiv_{\mathtt{W}}$.

For instance, on the graph in Figure 2.9, it is easy to check that applying summarization on $(\mathtt{G}_{/\mathtt{W}})^{\infty}$ (as prescribed by the shortcut) leads exactly to a graph strongly isomorphic to $(\mathtt{G}^{\infty})_{/\mathtt{W}}$.

Theorem 2 (S shortcut)**.** *The shortcut holds for* $\equiv_{\mathtt{S}}$.

Finally, we have:

Theorem 3 (No shortcut for $\equiv_{\mathtt{TW}}$)**.** *The shortcut does not hold for* $\equiv_{\mathtt{TW}}$.

We borrow from [4] Figure 2.10 which shows a counter-example. In G and $\mathtt{G}_{/\mathtt{TW}}$, all data nodes are untyped; only after saturation a node gains the type C. Thus, in $\mathtt{G}_{/\mathtt{TW}}$, one (untyped) node represents all data property subjects; this is exactly a "hasty fusion". In $(\mathtt{G}_{/\mathtt{TW}})^{\infty}$, this node gains a type, and in $((\mathtt{G}_{/\mathtt{TW}})^{\infty})_{/\mathtt{TW}}$, it is represented by a single node. In contrast, in $\mathtt{G}^{\infty}$, r_1 is typed and r_2 isn't, leading to two distinct nodes in $(\mathtt{G}^{\infty})_{/\mathtt{TW}}$. This is not isomorphic with $(\mathtt{G}_{/\mathtt{TW}})^{\infty}$ which, in this example, is strongly isomorphic to $((\mathtt{G}_{/\mathtt{TW}})^{\infty})_{/\mathtt{TW}}$. Thus, the shortcut does not hold for $\equiv_{/\mathtt{TW}}$ does not admit a shortcut.

The last two shortcut theorems established prior to my internship are:

Theorem 4 (No shortcut for $\equiv_{\mathtt{TS}}$)**.** *The shortcut does not hold for* $\equiv_{\mathtt{TS}}$.

The graph in Figure 2.10 is also a shortcut counter-example for TS. More generally, let $\equiv_{\mathtt{X}}$ be an arbitrary RDF node equivalence, and $\equiv_{\mathtt{TX}}$ be a type-first summary obtained by replacing in Definition 7, $\equiv_{\mathtt{W}}$ by $\equiv_{\mathtt{X}}$. Based on this counter-example, one can show that the shortcut does not hold for $\equiv_{\mathtt{TX}}$. If the ontology only features subclass triples, the shortcut holds also for $\equiv_{\mathtt{TW}}$ and $\equiv_{\mathtt{TS}}$; this is because any node typed in $\mathtt{G}^{\infty}$ was already typed in G.

Theorem 5. (BISIMILARITY SHORTCUT) *The shortcut holds for the forward* ($\equiv_{\mathtt{fb}}$)*, backward* ($\equiv_{\mathtt{bw}}$)*, and forward-and-backward* ($\equiv_{\mathtt{fb}}$) *bisimilarity equivalence relations (recalled in Section 2.2).*

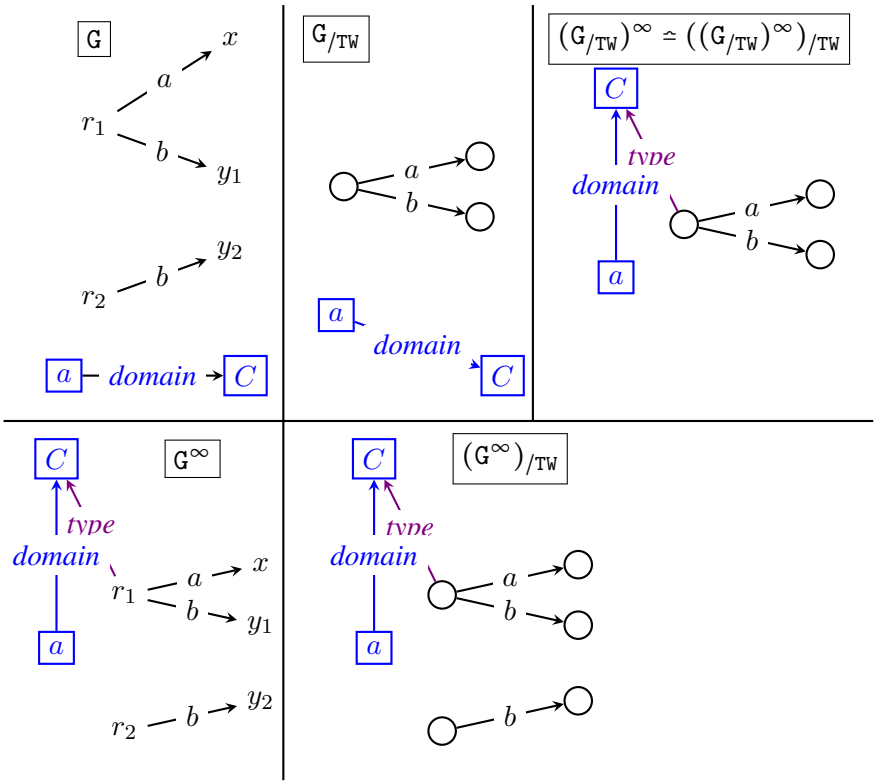


Figure 2.10: Shortcut counter-example.

Chapter 3

Summarization aware of type hierarchies

In this chapter we describe a new RDF graph summarization technique that takes into account the type hierarchy. Section 3.1 introduces this novel technique and provides the definition of new equivalence relation. Section 3.2 establishes the formal definition of the quotient summary based on that equivalence relation.

3.1 Novel type-based RDF equivalence

Our first goal is to *define a equivalence relation* which:

1. takes type information into account, thus belongs to the types-then-data approach;
2. leads to a quotient summary which represents together, to the extent possible (see below), nodes that have *the same most general type*.

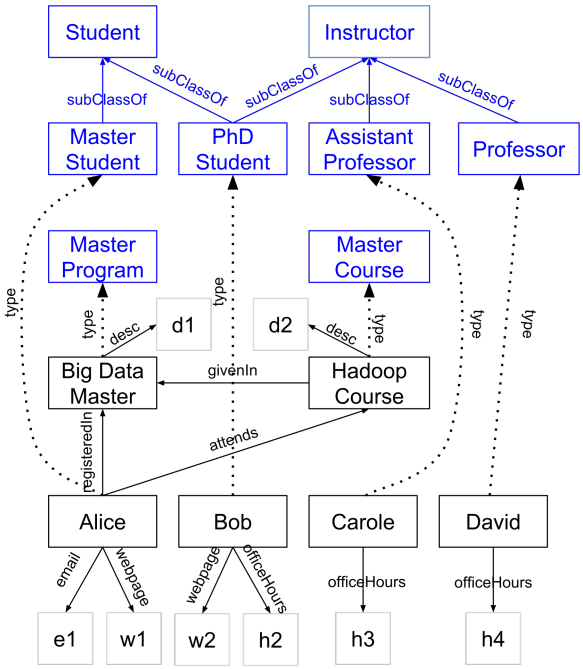


Figure 3.1: Sample RDF graph.

Formally, let $\mathcal{C} = \{c_1, c_2, \ldots, \}$ be the set of class nodes present in G (that is, in SG and/or in TG). We can view these nodes as organized in a *directed graph* where there is an edge $c_1 \rightarrow c_2$ as long as

G's saturated schema $\mathtt{SG}^\infty$ states that c_1 is a subclass of c_2. By a slight abuse of notation, we use $\mathcal{C}$ to also refer to this graph[1]. In principle, $\mathcal{C}$ could have cycles, but this does not appear to correspond to meaningful schema designs. Therefore, we assume without loss of generality that $\mathcal{C}$ is a directed acyclic graph (DAG, in short)[2]. In Figure 3.1, $\mathcal{C}$ is the DAG comprising the eight (blue) class nodes and edges between them; this DAG has four roots.

First, assume that $\mathcal{C}$ is a tree, e.g., with *Instructor* as a root type and *PhDStudent*, *AssistantProfessor* as its subclasses. In such a case, we would like instances of all the abovementioned types to be represented together, because they are all instances of the top type *Instructor*. This extends easily to the case when $\mathcal{C}$ is a forest, e.g., a second type hierarchy in $\mathcal{C}$ could feature a root type *Paper* whose subclasses are *ConferencePaper*, *JournalPaper* etc. In this case, we aim to represent all authors together because they are instances of *Paper*.

In general, though, $\mathcal{C}$ may not be a forest, but instead it may be a graph where some classes have multiple superclasses, potentially unrelated. For instance, in Figure 3.1, *PhDStudent* has two superclasses, *Student* and *Instructor*. Therefore, it is not possible to represent G nodes of type *PhDStudent* based on their most general type, because they have more than one such type. Representing them twice (once as *Instructor*, once as *Student*) would violate the framework (Definition 2.2), in which any summary is a quotient and thus, each G node must be represented by exactly one summary node.

To represent resources as much as possible according to their most general type, we proceed as follows.

Definition 11. (TREE COVER) *Given a DAG $\mathcal{C}$, we call a* tree cover *of $\mathcal{C}$ a set of trees such that: (i) each node in $\mathcal{C}$ appears in exactly one tree; (ii) together, they contain all the nodes of $\mathcal{C}$; and (iii) each $\mathcal{C}$ edge appears either in one tree or connects the root of one tree to a node in another.*

Given $\mathcal{C}$ admits many tree covers, however, it can be shown that there exists a tree cover with the least possible number of trees, which we will call **min-size cover**. This cover can be computed in a single traversal of the graph by *creating a tree root exactly from each $\mathcal{C}$ node having two supertypes such that none is a supertype of the other*, and attaching to it all its descendants which are not themselves roots of another tree. For instance, the RDF schema from Figure 3.1 leads to a min-size cover of five trees:

- A tree rooted at *Instructor* and the edges connecting it to its children *AssistantProfessor* and *Professor*;
- A single-node tree rooted at *PhDStudent*;
- A tree rooted at *Student* with its child *MasterStudent*;
- A single-node tree for *MasterProgram* and another for *MasterCourse*.

Figure 3.2 illustrates min-size covers on a more complex RDF schema, consisting of the types A to Q. Every arrow goes from a type to one of its supertypes (for readability, the figure does not include all the implicit subclass relationships, e.g., that E is also a subclass of H, I, J etc.). The pink areas each denote a tree in the corresponding min-size cover. H and L are tree roots because they have multiple, unrelated supertypes.

To complete our proposal, we need to make an extra hypothesis on G:

(†) Whenever a data node n is of two distinct types c_1, c_2 which are *not in the same tree in the min-size tree cover* of $\mathcal{C}$, then (i) c_1 and c_2 have some common subclasses, (ii) among these, there exists a class $c_{1,2}$ that is a superclass of all the others, and (iii) n is of type $c_{1,2}$.

[1]Ontology languages such as RDF Schema or OWL feature a top type, that is a supertype of any other type, such as rdfs:Resource. We do not include such a generic, top type in $\mathcal{C}$.

[2]If $\mathcal{C}$ has cycles, the types in each cycle can all be seen as equivalent, as each is a specialization of all the other, and could be replaced by a single (new) type in a simplified ontology. The process can be repeated until $\mathcal{C}$ becomes a DAG, then the approach below can be applied, following which the simplified types can be restored, replacing the ones we introduced. We omit the details.

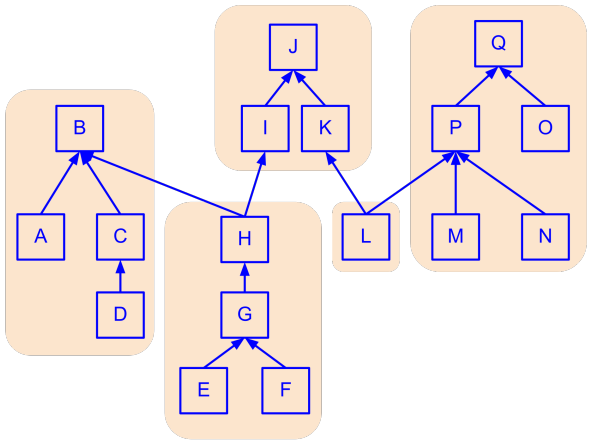


Figure 3.2: Sample RDF schema and min-size cover of the corresponding $\mathcal{C}$.

For instance, in our example, hypothesis (†) states that if a node n is an *Instructor* and a *Student*, these two types must have a common subclass (in our case, this is *PhDStudent*), and n must be of type *PhDStudent*. The hypothesis would be violated if there was another common subclass of *Instructor* and *Student*, say *MusicLover*[3], that was neither a subclass of *PhDStudent* nor a superclass of it.

(†) may be checked by a SPARQL query on G. While it may not hold, we have not found such counter-examples in a set of RDF graphs we have examined (see Section 4.2). In particular, (†) immediately holds in the frequent case when $\mathcal{C}$ is a tree (taxonomy) or, more generally, a forest: in such cases, the min-size cover of $\mathcal{C}$ is exactly its set of trees, and any types c_1, c_2 of a data node n are in the same tree.

When (†) holds, we can state:

Lemma 1 (Lowest branching type)**.** *Let* G *be an RDF graph satisfying* (†)*, n be a data node in* G*, cs_n be the set of types of n in* G*, and cs_n^∞ be the classes from cs_n together with all their superclasses (according to the saturated schema of* G*). Assume that $cs_n^\infty \neq \emptyset$.*

Then there exists a type lbt_n, called lowest branching type*, such that:*

- *$cs_n^\infty = cs'_n \cup cs''_n$, where $\{lbt_n\} \in cs'_n$ and cs''_n may be empty;*
- *the types in cs'_n (if any) can be arranged in a tree according to* subclass *relation between them, and the most general one is lbt_n;*
- *if cs''_n is not empty, it is at least of size two, and all its types are superclasses of lbt_n.*

Proof: Let's assume to the contrary that there exists an RDF graph $\mathtt{G}_1$ satisfying (†), a node n in $\mathtt{G}_1$, cs_n the set of types of n, $cs_n^\infty \neq \emptyset$ is the set of types of n with all their supertypes (according to saturated schema of $\mathtt{G}_1$) and there is no *lowest branching type* for n.

Let $\mathcal{G}$ be the set of all such RDF graphs and let G be the $\mathcal{G}$ graph containing a node n that violates the lemma and such that $|cs_n^\infty|$ is the smallest, across all such lemma-violating nodes n in any graph from $\mathcal{G}$.

Let $k = |cs_n^\infty|$. Note that $k > 0$ by definition. Let's consider the cases:

1. $k = 1$ In this case, the lemma trivially holds.
2. $k \geq 2$ In this case, let $t_1, \ldots, t_k$ be the types of node n (their order not important). Let's consider graph $\mathtt{G}'$ which is the same as G but without node n having type t_k. From the way we chose G and $\mathtt{G}'$, $\mathtt{G}'$ satisfies the lemma, thus there exists a *lowest branching type* lbt_n for n in $\mathtt{G}'$. Now, let's add t_k to the types of n in $\mathtt{G}'$. There are 3 possibilities:

[3] *MusicLover* may be a subclass of yet another class (distinct type c_3 in third other *min-size tree*) and it would still violate the hypothesis

(a) t_k is a subclass of lbt_n. Then lbt_n is also *lowest branching type* after this addition.

(b) t_k is a superclass of lbt_n. If it's the only superclass of lbt_n then t_k is a new *lowest branching type*, else n still admits the lowest branching type lbt_n.

(c) t_k is neither a sub- nor a superclass of lbt_n. Then it is in another tree in min-size cover of G, thus by (†) it follows that t_k and some other type between $t_1, \ldots, t_{k-1}$ have a common subtype which serves as a lowest branching type for n.

From the above discussion we conclude that the node n for which $k = |cs_n^\infty|$ is not the lemma counterexample with the smallest k, which contradicts the assumption we made when picking it! Therefore no graph exists in $\mathcal{G}$, thus all Gs satisfy the lemma. □

For instance, let n be Bob in Figure 3.1, then cs_n is {*PhDStudent*}, thus cs_n^∞ is {*PhDStudent, Student, Instructor*}. In this case, lbt_n is *PhDStudent*, cs'_n is {*PhDStudent*} and cs''_n is {*Student, Instructor*}.

If we take n to be Carole, cs_n^∞ is {*AssistantProfessor, Instructor*}; no type from this set has two distinct superclasses, thus cs''_n must be empty, lbt_{Carole} is *Instructor*, and cs'_n is {*AssistantProfessor*, *Instructor*}. By a similar reasoning, lbt_{David} is *Instructor*, and lbt_{Alice} is *Student*. When n has a type without subclasses or superclasses, such as BigDataMaster, it leads to cs''_n being empty, and cs'_n is lbt_n, the only type of n. Thus, $lbt_{\text{BigDataMaster}}$ is *MasterProgram* and $lbt_{\text{HadoopCourse}}$ is *MasterCourse*.

For a more complex example, recall the RDF schema in Figure 3.2, and let n be a node of type E in an RDF graph having this schema. In this case, cs_n is $\{E, G, H, B, I, J\}$, lbt_n is H, cs'_n is $\{E, G, H\}$ while cs''_n is $\{B, I, J\}$. Based on Lemma 1, we define our novel notion of equivalence, reflecting the hierarchy among the types of G data nodes:

Definition 12. (TYPE-HIERARCHY EQUIVALENCE) Type-hierarchy equivalence*, denoted* $\equiv_{\text{TH}}$*, is an equivalence relation defined as follows: two data nodes* n_1 *and* n_2 *are type-hierarchy equivalent, noted* $n_1 \equiv_{\text{TH}} n_2$*, iff* $lbt_{n_1} = lbt_{n_2}$.

From the above discussion, it follows that Carole $\equiv_{\text{TH}}$ David, matching the intuition that they are both instructors and do not belong to other type hierarchies. In contrast, PhD students (such as Bob) are only type-hierarchy equivalent to each other; they are set apart by their dual *Student* and *Instructor* status. Master students such as Alice are only type-hierarchy equivalent among themselves, as they only belong to the student type hierarchy. Every other typed node of G is only type-hierarchy equivalent to itself.

3.2 RDF summary based on type hierarchy equivalence

Based on $\equiv_{\text{TH}}$ defined above, and the $\equiv_{\text{UW}}$ structural equivalence relation (two nodes are $\equiv_{\text{UW}}$ if they have no types, and are weakly equivalent), we introduce a novel summary belonging to the "type-first" approach:

Definition 13. (WEAK TYPE-HIERARCHY SUMMARY) *The type hierarchy summary of* G*, denoted* $\text{G}_{/\text{WTH}}$*, is the summary through* $\equiv_{\text{UW}}$ *of the summary through* $\equiv_{\text{TH}}$ *of* G*:*

$$\text{G}_{/\text{WTH}} = (\text{G}_{/\equiv_{\text{TH}}})_{/\equiv_{\text{UW}}}$$

Figure 3.4 illustrates the $\text{G}_{/\text{WTH}}$ summary of the RDF graph in Figure 3.1. Different from the weak summary (Figure 3.3), it does not represent together nodes of unrelated types, such as BigDataMaster and HadoopCourse. At the same time, different from the typed weak summary of the same graph, it does not represent separately each individual, and instead it keeps Carole and David together as they only belong to the instructor type hierarchy.

More summaries based on $\equiv_{\text{TH}}$ could be obtained by replacing UW with another RDF equivalence relation. For instance we can easily define strong counterpart of the weak type-hierarchy summary as follows:

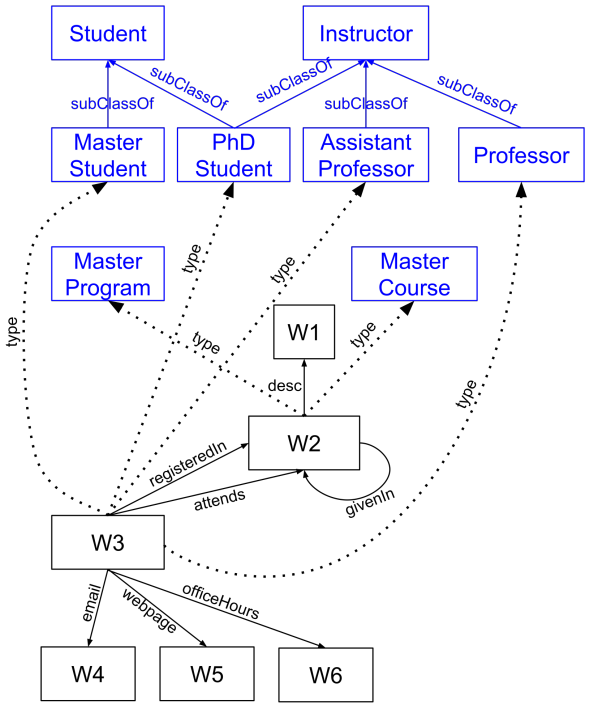


Figure 3.3: Weak summary of the sample RDF graph in Figure 3.1.

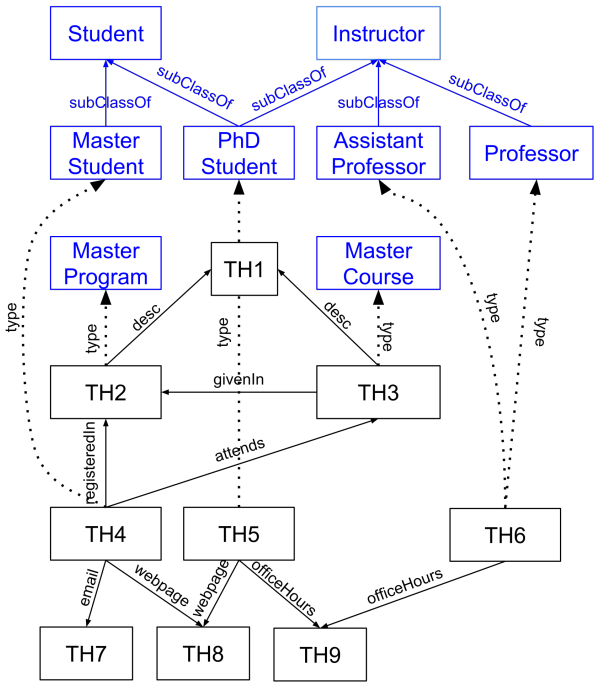


Figure 3.4: Weak type-hierarchy summary of the RDF graph in Figure 3.1. The roots of the trees in the min-size cover of $\mathcal{C}$ are underlined.

Definition 14. (STRONG TYPE-HIERARCHY SUMMARY) *The type hierarchy summary of* $\mathtt{G}$*, denoted* $\mathtt{G}_{/\mathtt{STH}}$*, is the summary through* $\equiv_{\mathtt{US}}$ *of the summary through* $\equiv_{\mathtt{TH}}$ *of* $\mathtt{G}$*:*

$$\mathtt{G}_{/\mathtt{STH}} = (\mathtt{G}_{/\equiv_{\mathtt{TH}}})_{/\equiv_{\mathtt{US}}}$$

Chapter 4

Graph summarization algorithms

This chapter focuses on the core contributions of my internship, which are of algorithmic nature, it describes the algorithms capable of constructing graph summaries out of RDF graphs. Section 4.1 considers centralized algorithms. The development of these was started by my supervisor and I have continued it. Section 4.2 outlines an algorithm for type hierarchy-based summarization. Section 4.3 presents distributed algorithms implemented in the Spark framework using Scala API. I am co-author of all the algorithms and co-designer of the implementation. My contribution to the global and distributed variants is major. Conversely, for the incremental version, the core concept was implemented by my supervisor and then I have finalized the work by covering all the corner cases and transforming the processing of the graphs into the more general form.

4.1 Centralized summarization algorithms

We now discuss summarization algorithms which, given as input a graph $\mathtt{G}$, construct $\mathtt{G}_{/\mathtt{W}}$, $\mathtt{G}_{/\mathtt{S}}$, $\mathtt{G}_{/\mathtt{TW}}$ and $\mathtt{G}_{/\mathtt{TS}}$. The schema $\mathtt{S}_\mathtt{G}$ is directly copied in the summary thus, below we focus on summarizing $\mathtt{D}_\mathtt{G}$ (Section 4.1.1) and $\mathtt{T}_\mathtt{G}$ (Section 4.1.2).

4.1.1 Data graph summarization

We have devised summarization algorithms of two flavors.

Ⓐ **Global** algorithms start by learning the equivalence relation[1] and creating the summary nodes. Then, a final pass over $\mathtt{G}$ computes $f_\equiv$ and adds to the summary the edge $f_\equiv(n_1)\ e\ f_\equiv(n_2)$ for each triple $n_1\ e\ n_2$ in $\mathtt{G}$. While this requires several passes over the graph, it connects its nodes directly to their final representatives they will have in $\mathtt{G}_{/\equiv}$.

We start with our **global $\mathtt{W}$ summarization algorithm** (Table 4.1). It exploits Proposition 1, which guarantees that any data property occurs only once in the summary. To each distinct data property $\mathtt{p}$ encountered in $\mathtt{D}_\mathtt{G}$, it associates a summary node (integer) $s_\mathtt{p}$ which will be the (unique) source of $\mathtt{p}$ in the summary, and similarly a node $t_\mathtt{p}$ target of $\mathtt{p}$; these are initially unknown, and evolve as $\mathtt{G}$ is traversed. Further, it uses two maps op and ip which associate to each $\mathtt{D}_\mathtt{G}$ node n, the set of its outgoing, resp. incoming data properties. These are filled during the first traversal of $\mathtt{D}_\mathtt{G}$ (step 1.) Steps 2. to 2.5 ensure that for each node n having outgoing properties and possibly incoming ones, $s_\mathtt{p}$ for all the outgoing ones are equal, and equal also to $t_\mathtt{p}$ for all the incoming ones. This is performed using a function fuse which, given a set of summary nodes, picks one that will replace all of them. In our implementation, summary nodes are integers, and fuse is simply min; we just need fuse to be distributive over $\cup$, i.e., $\mathrm{fuse}(A, (B \cup C)) = \mathrm{fuse}(\mathrm{fuse}(A, B), \mathrm{fuse}(A, C))$. Symmetrically, step 3. ensures that the incoming properties of nodes lacking outgoing properties (thus, absent from op) also have the same

[1] Recall that $\equiv_\mathtt{W}$, $\equiv_\mathtt{S}$, $\equiv_\mathtt{T}$, as well as bisimilarity equivalence, are defined *based on the data/type triples of a given graph* $\mathtt{G}$, thus when starting to summarize $\mathtt{G}$, we do not know whether any two nodes are equivalent; the full $\equiv$ is known only after inspecting all $\mathtt{G}$ triples.

Algorithm **global-W(G)**

1. For each $\mathtt{s}\ \mathtt{p}\ \mathtt{o} \in \mathtt{G}$, add $\mathtt{p}$ to $op(\mathtt{s})$ and to $ip(\mathtt{o})$.
2. For each node $n \in op$:
2.1. Let $X \leftarrow \text{fuse}\{s_\mathtt{p} \,|\, \mathtt{p} \in op(n)\}$.
If X is undefined, let $X \leftarrow$nextNode();
2.2. Let $Y \leftarrow \text{fuse}\{t_\mathtt{p} \,|\, \mathtt{p} \in ip(n)\}$.
If Y is undefined, let $Y \leftarrow$nextNode();
2.3. Let $Z \leftarrow \text{fuse}(X, Y)$;
2.4. For each $\mathtt{p} \in ip(n)$, let $s_\mathtt{p} \leftarrow Z$;
2.5. For each $\mathtt{p} \in op(n)$, let $t_\mathtt{p} \leftarrow Z$;
3. Repeat **2** to **2.5** swapping ip with op and $t_\mathtt{p}$ with $s_\mathtt{p}$;
4. For each $\mathtt{s}\ \mathtt{p}\ \mathtt{o} \in \mathtt{G}$: let $f_\mathtt{W}(\mathtt{s}) \leftarrow s_\mathtt{p}$, $f_\mathtt{W}(\mathtt{o}) \leftarrow t_\mathtt{p}$;
Add $f_\mathtt{W}(\mathtt{s})\ \mathtt{p}\ f_\mathtt{W}(\mathtt{o})$ to $\mathtt{G}_{/\mathtt{W}}$.

Table 4.1: Global W summarization algorithm.

Algorithm **global-S(G)**

1. For each $\mathtt{s}\ \mathtt{p}\ \mathtt{o} \in \mathtt{G}$:
1.1. Check if $src_\mathtt{p}$, $trg_\mathtt{p}$, $sc(\mathtt{s})$ and $tc(\mathtt{o})$ are known; those not known are initialized with $\{\mathtt{p}\}$;
1.2. If $sc(\mathtt{s}) \neq src_\mathtt{p}$, fuse them into new clique $src'_\mathtt{p} = sc(\mathtt{s}) \cup src_\mathtt{p}$; similarly, if $tc(\mathtt{o}) \neq trg_\mathtt{p}$, fuse them into $trg'_\mathtt{p} = tc(\mathtt{o}) \cup trg_\mathtt{p}$. **2.** For each $\mathtt{s}\ \mathtt{p}\ \mathtt{o} \in \mathtt{G}$:
2.1. $f_\mathtt{S}(\mathtt{s}) \leftarrow$ the (unique) summary node corresponding to the cliques $(sc(\mathtt{s}), tc(\mathtt{s}))$; similarly, $f_\mathtt{S}(\mathtt{o}) \leftarrow$ the node corresponding to $(sc(\mathtt{o}), tc(\mathtt{o}))$ (create the nodes if needed).
2.2 Add $f_\mathtt{S}(\mathtt{s})\ \mathtt{p}\ f_\mathtt{S}(\mathtt{o})$ to $\mathtt{G}_{/\mathtt{S}}$.

Table 4.2: Global S summarization algorithm.

target. In Step 4., we represent s and o based on the source/target of the property p connecting them. The fuse operations in 2. and 3. have ensured that, while traversing G triples in 4., a same G node n is always represented by the same summary node $f_\mathtt{W}(n)$.

Our **global S summarization algorithm** (Table 4.2) uses two maps sc and tc which store for each data node $n \in \mathtt{D_G}$, its source clique $sc(n)$, and its target clique $tc(n)$, and for each data property p, its source clique $src_\mathtt{p}$ and target clique $trg_\mathtt{p}$. Further, to each (source clique, target clique) pair encountered until a certain point during summarization, we store the (unique) corresponding summary node. Steps 1.-1.2. build the source and property cliques present in G and associate them to every subject and object node (in sc and tc), as well as to any data property (in $src_\mathtt{p}$ and $trg_\mathtt{p}$). For instance, on the sample graph in Figure 2.2, these steps build the cliques in Table 2.1. Steps 2-2.2. represent the nodes and edges of G.

The correctness of algorithms **global-W** and **global-S** follows quite easily from their descriptions and the summary definitions.

(B) Incremental algorithms simultaneously learn the equivalence relation from G *and* represent G data triples. They are particularly suited for incremental **summary maintenance**: if new triples $\Delta^+_\mathtt{G}$ are added to G, it suffices to run the summarization algorithm only on $\Delta^+_\mathtt{G}$, based on $\mathtt{G}_\equiv$ and its representation function $f_\equiv$, in order to obtain $(\mathtt{G} \cup \Delta^+_\mathtt{G})_{/\equiv}$. Incremental algorithms also provide the basic building blocks for incrementally propagating the effect of a deletion from G. However, incremental algorithms are considerably **more complex**, since various decisions (assigning sources/targets to properties in W, source/target cliques in S, node representatives in both) must be *repeatedly revisited* to reflect newly acquired knowledge. We illustrate this on our algorithms and examples below.

Each **incremental summarization** algorithm consists of an **incremental update method, called for every $\mathtt{D_G}$ triple**, which adjusts the summary's data structures, so that at any point, the summary reflects exactly the graph edges (triples) visited until then.

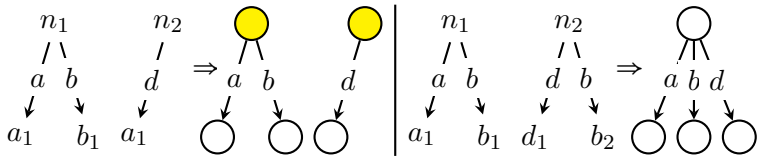
Table 4.3 outlines incremental W summarization. For example (see the figure below), assume the

Algorithm **increm-W(s p o)**

1. Check if $s_\mathtt{p}$ and $o_\mathtt{p}$ are known: either both are known (if a triple with property p has already been traversed), or none;
2. Check if $f_\mathtt{W}(\mathtt{s})$ and $f_\mathtt{W}(\mathtt{o})$ are known; none, one, or both may be, depending on whether s, respectively o have been previously encountered;
3. Fuse $s_\mathtt{p}$ with $f_\mathtt{W}(\mathtt{s})$ (if one is unknown, assign it the value of the other), and $o_\mathtt{p}$ with $f_\mathtt{W}(\mathtt{o})$;
4. Update $f_\mathtt{W}(\mathtt{s})$ and $f_\mathtt{W}(\mathtt{o})$, if needed;
5. Add the edge $f_\mathtt{W}(\mathtt{s})$ p $f_\mathtt{W}(\mathtt{o})$ to $\mathtt{G}_{/\mathtt{W}}$.

Table 4.3: Incremental W summarization of one triple.

algorithm traverses the graph G in Figure 2.2 starting with: $n_1\ a\ a_1$, then $n_1\ b\ b_1$, then $n_2\ d\ d_1$. When we summarize this third triple, we do not know yet that the source of a d triple is also equivalent to n_1, because no common source of b and d (e.g., n_2 or n_4) has been seen so far. Thus, n_2 is found not equivalent to any node visited so far, and represented separately from n_1. Now assume the fourth triple traversed is $n_2\ b\ b_2$: at this point, we know that a, b and d are in the same source clique, thus $n_1 \equiv_\mathtt{W} n_2$, and their representatives (highlighted in yellow below) must be **fused** in the summary (Step 3.) More generally, it can be shown that $\equiv_\mathtt{W}$ **only grows** as more triples are visited, in other words: if in a subset $\mathtt{G}'$ of G's triples, two nodes n_1, n_2 are weakly equivalent, then this holds in any $\mathtt{G}''$ with $\mathtt{G}' \subseteq \mathtt{G}'' \subseteq \mathtt{G}$.



Summary node **fusion** dominates the algorithm's **complexity**. Let N_1, N_2 be two sets of G nodes, represented at a certain point by the distinct summary nodes m_1, m_2. When fusing the latter into a single m, we must also record that all the nodes in $N_1 \cup N_2$ are now represented by m. A naïve implementation leads to $O(N^2)$ complexity, where N is the number of nodes in $\mathtt{D}_\mathtt{G}$, since each new node may lead to a fusion whose cost is $O(N)$; in the worst case N could be proportional to $|\mathtt{G}|$, the number of triples in G, leading to an overall complexity of $O(|\mathtt{G}|^2)$ for the incremental weak summarization.

Instead, we rely on a **Union-Find** (aka Disjoint Sets) data structure, with the *path compression* and *union by size*[2] optimizations, which guarantee an overall **quasi-linear worst-case complexity** to our incremental weak summarization algorithm. The exact complexity is $O(N\alpha(N))$ where $\alpha(N)$, the inverse Ackermann's function, is smaller than 5 for any machine-representable input N. Assimilating this to **linear-time**, the algorithm's complexity class is in $O(|\mathtt{G}|)$, which is also **optimal**, as summarization cannot do less than fully traversing G.

Table 4.4 outlines the incremental update of the S summary due to the traversal of the triple s p o. Conceptually, the algorithm is symetric for the source (s) and target (o) of the edge, we only discuss the source side below. Steps 1. and 2. start by determining the source clique of s, based on its previously known source clique (if any) and the previously known source clique of p (if any); after step 2., s's source (and target) clique reflecting also the newly seen triple s p o are completely known. Determining them may have involved fusing some previously separate cliques. For instance, on the graph in Figure 2.2, assume we first traverse the two a triples, then we traverse $n_2\ b\ b_2$; so far we have the source cliques $\{a\}$, $\{b\}$ and $\varnothing$. If the next traversed triple is $n_2\ a\ a_2$, we fuse the source cliques (step 3.1) $\{a\}$ and $\{b\}$ into $\{a, b\}$. This requires fusing the summary node whose (source, target) cliques were ($\{a\}$, $\varnothing$) with the one which had ($\{b\}$, $\varnothing$) (Step 3.2).

The last intricacy of incremental strong summarization is due to the fact that unlike $\equiv_\mathtt{W}$, $\equiv_\mathtt{S}$ **may grow and shrink** during incremental, strong summarization. For instance, assume incremental strong

[2] https://en.wikipedia.org/wiki/Disjoint-set_data_structure

Algorithm **increm-S(s p o)**

1. Check if we already know a source clique $src_{\mathtt{p}}$ (resp. target clique $trg_{\mathtt{p}}$). Either both are known (if a p triple has already been traversed), or none. Those not known are initialized with $\{\mathtt{p}\}$;
2. Check if $sc(\mathtt{s})$ (resp. $tc(\mathtt{o})$) are known; those unknown are initialized with $\{\mathtt{p}\}$;
3. If $sc(\mathtt{s}) \neq src_{\mathtt{p}}$, fuse them into new clique $src'_{\mathtt{p}} = sc(\mathtt{s}) \cup src_{\mathtt{p}}$, using Union-Find; similarly, if $tc(\mathtt{o}) \neq trg_{\mathtt{p}}$, fuse them into $trg'_{\mathtt{p}} = tc(\mathtt{o}) \cup trg_{\mathtt{p}}$, and:
3.1 Replace $sc(\mathtt{s})$ and $src_{\mathtt{p}}$ with $src'_{\mathtt{p}}$ throughout the summary (respectively, replace $tc(\mathtt{o})$ and $trg_{\mathtt{p}}$ with $trg'_{\mathtt{p}}$);
3.2 The above may entail summary node fusions; in this case, update $f_{\mathtt{S}}$ (use Union-Find) and the summary edges to reflect it;
4. If before seeing s p o s had been already represented and it had an empty source clique, then s needs to split, i.e., be represented separately from the nodes to which it was $\equiv_{\mathtt{S}}$ previously; call **split-source(s)**. (Symmetric discussion for o, call **split-target(o)**).
5. Update $f_{\mathtt{S}}(\mathtt{s})$ and $f_{\mathtt{S}}(\mathtt{o})$, if needed;
6. Add the edge $f_{\mathtt{S}}(\mathtt{s})$ p $f_{\mathtt{S}}(\mathtt{o})$ to $\mathtt{G}_{/\mathtt{S}}$.

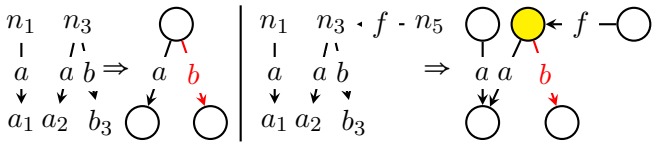
Table 4.4: Incremental S summarization of one triple.

Procedure **split-source(s)**

1. Collect all G edges adjacent to s into $transfer$ set.
2. For each s p o $\in transfer$, decrement by 1 the counter for $f_{\mathtt{S}}(\mathtt{s})$ p $f_{\mathtt{S}}(\mathtt{o})$ in the summary.
3. Update $f_{\mathtt{S}}(\mathtt{s})$.
4. For each s p o $\in transfer$, if such edge already exists in the summary, then increment its counter by 1, otherwise add $f_{\mathtt{S}}(\mathtt{s})$ p $f_{\mathtt{S}}(\mathtt{o})$ to the summary with counter equal to 1.

Table 4.5: Procedure of splitting summary node s on source.

summarization of the graph in Figure 2.2 starts with $n_1\ a\ a_1$, $n_3\ a\ a_2$, $n_3\ b\ b_3$ (see the figure below). After these, we know $n_1 \equiv_{\mathtt{S}} n_3$; their source clique is $\{a, b\}$ and their target clique is $\varnothing$. Assume the next triple traversed is $n_5\ f\ n_3$: at this point, n_3 is *not* $\equiv_{\mathtt{S}}$ *to* n_1 *any more*, because n_5's target clique is now $\{f\}$ instead of the empty $\varnothing$. Thus, n_5 **splits** from n_1, that is, it needs to be represented by a new summary node (shown in yellow below), distinct from the representative of n_1.



Further, note that the representative of n_1 and n_3 (at left above) had one b edge (highlighted in red) which was *solely due to* n_3*'s outgoing* b *edge*. By definition of a quotient summary (Section 2.2), that edge *moves* from the old to the new representative of n_3 (the yellow node). If, above at left, n_1 had also had an outgoing edge labeled b, at right, both nodes in the top row would have had an outgoing b edge. It can be shown that *splits only occur in such cases*, i.e., o whose target clique becomes non-empty (respectively, s whose source clique becomes non-empty, and the node was previously represented together with other nodes; if it was represented alone, we just update the respective clique of its representative).

The procedure **split-source(s)** (4.5) represents s separately, to reflect it no longer has an empty target clique, and, for each outgoing edge of s: adds a corresponding edge to the new representative of s; and checks if, as a consequence, an edge needs to be removed from its previous representative.

Proposition 4. (ALGORITHM CORRECTNESS) *Applying algorithm* **increm-W** *(respectively,* **increm-S***) successively on each triple of* G*, in any order, builds* $\mathtt{G}_{/\mathtt{W}}$ *(respectively,* $\mathtt{G}_{/\mathtt{S}}$*).*

Splitting requires inspecting the data edges attached to the splitting node, in order to add to its new representative the edges it must have (such as the $n_3\ b\ b_3$ above). We make the hypothesis, denoted ($\bigstar$), that the maximum number of edges incoming/outgoing a data node is small (and basically constant) compared to the size of G; this was the case in the graphs we have experimented with. To keep splitting cost under control: (i) We store for each summary node m and edge e a *counter* $m_\sharp, e_\sharp$ of the $\mathtt{D_G}$ nodes and edges it represents. Splitting is only needed when $m_\sharp$>1. (ii) Summary node m loses an outgoing (resp. incoming) edge e labeled p when s (resp. o) splits, if and only if the number of outgoing s edges (resp. incoming o edges) labeled p equals $e_\sharp$. At left in the figure above, $e_\sharp$ was 1, thus the b edge is removed from the old representative of n_3.

Under the ($\bigstar$) hypothesis, using the data structures (including Union-Find) described above, **the complexity of incremental strong summarization is amortised constant per added triple**.

All our algorithms require $O(|\mathtt{G}|)$ space to store the summary edges, the representation function, and their other data structures.

4.1.2 Typed graph summarization

We now explain how to extend our incremental $\mathtt{D_G}$ summarization algorithms to type triples.

To extend W, respectively, S summarization to type triples in "data-then-type" fashion (Section 2.4.1), we run W, resp. S summarization *first, over* $\mathtt{D_G}$ *triples only*, as described in Section 4.1.1. This assigns their (final) representatives to all nodes of $\mathtt{D_G}$. Then, for each s *type* C triple, we simply add to the summary the edge $f_\mathtt{W}(\mathtt{s})$ *type* C (resp. $f_\mathtt{S}(\mathtt{s})$ *type* C); recall from Section 2.4 that any class node C is represented by itself.

For "type-then-data" summarization (Section 2.4.2), that is, for TW and TS, *we first traverse* $\mathtt{T_G}$ *triples only*, compute all the class sets, and assign to each typed data node a representative based on its class set. Then, we run a *type-aware variant* of a W (resp. S) algorithm, either global or incremental. The changes introduced in the type-aware variant are: (i) In TW summarization, a data property p may lack an untyped source, if p only occurs on typed nodes; consider for instance the graph whose only triples are n_1 *type* C, $n_1\ e\ a_1$. Similarly, in TS summarization, a property (e.g., e above) may have a target clique, but lack a source clique, since it does not have an untyped source. (ii) Summarizing the data triple s p o does not fuse nor split the representative of s (resp. o) if s (resp. *object*) is typed; instead, representatives of typed nodes (computed during the *type* triples traversal) are left untouched.

Proposition 5. (ALGORITM CORRECTNESS) *Applying global-*W *(respectively global-*S*) on* G*, or applying increm-*W *(respectively, increm-*S*) on each triple of* G*, extended as described above for data-then-type or type-then-data summarization leads, respectively* $\mathtt{G_{/W}}$, $\mathtt{G_{/S}}$, $\mathtt{G_{/TW}}$ *and* $\mathtt{G_{/TS}}$.

These algorithms need to work with $\mathtt{S_G}$, $\mathtt{D_G}$ and $\mathtt{T_G}$ *separately*. Fortunately, most popular RDF store allow such direct access. The space needed to also represent $\mathtt{T_G}$ triples remains linear in $|\mathtt{G}|$.

4.2 Type-hierarchy-based summarization algorithms

4.2.1 Constructing the weak type-hierarchy summary

An algorithm which builds $\mathtt{G_{/WTH}}$ is as follows:

1. From SG, build $\mathcal{C}$ and its min-size cover.
2. For every typed node n of G, identify its lowest branching type lbt_n and (the first time a given lbt_n is encountered) create a new URI URI_{lbt_n}: this will be the $\mathtt{G_{/WTH}}$ node representing all the typed G nodes having the same lbt_n.
3. Build the weak summary of the untyped nodes of G, using the algorithm described in [7]. This creates the untyped nodes in $\mathtt{G_{/WTH}}$ and all the triples connecting them.

4. *Add type edges:* for every triple n *type* c in $\mathtt{G}$, add (unless already in the summary) the triple URI_{lbt_n} *type* c to $\mathtt{G}_{/\mathtt{WTH}}$.

5. *Connect the typed and untyped summary nodes:* for every triple $n_1\ p\ n_2$ in $\mathtt{G}$ such that n_1 has types in $\mathtt{G}$ and n_2 does not, add (unless already in the summary) the triple $URI_{lbt_{n_1}}\ p\ UW_{n_2}$ to $\mathtt{G}_{/\mathtt{WTH}}$, where UW_{n_2} is the node representing n_2, in the weak summary of the untyped part of $\mathtt{G}$. Apply a similar procedure for the converse case (when n_1 has no types but n_2 does).

Step 1) is the fastest as it applies on the schema, typically orders of magnitude smaller than the data. The cost of the steps 2)-4) depend on the distribution of nodes (typed or untyped) and triples (type triples; data triples between typed/untyped nodes) in $\mathtt{G}$. [7] presents an efficient, almost-linear time (in the size of $\mathtt{G}$) weak summarization algorithm (step 3). The complexity of the other steps is linear in the number of triples in $\mathtt{G}$, leading to an overall almost-linear complexity.

4.2.2 Applicability

To understand if $\mathtt{G}_{/\mathtt{WTH}}$ summarization is helpful for an RDF graph, the following questions should be answered:

1. Does $\mathtt{SG}$ feature subclass hierarchies? If it does not, then $\mathtt{G}_{/\mathtt{WTH}}$ reduces to the weak summary $\mathtt{G}_{\mathtt{TW}}$.

2. Does $\mathtt{SG}$ feature a class with two unrelated superclasses?

   (a) No: then $\mathcal{C}$ is a tree or a forest. In this case, $\mathtt{G}_{/\mathtt{WTH}}$ represents every typed node together with all the nodes whose type belong to the same type hierarchy (tree).

   (b) Yes: then, does $\mathtt{G}$ satisfy (†)?

      i. Yes: one can build $\mathtt{G}_{/\mathtt{WTH}}$ to obtain a refined representation of nodes according to the lowest branching type in their type hierarchy.

      ii. No: $\mathtt{G}_{/\mathtt{WTH}}$ is undefined, due to the lack of a unique representative for the node(s) violating (†).

Among the RDF datasets frequently used, DBLP[3], the BSBM benchmark [2], and the real-life Slegger ontology[4] whose description has been recently published [24] exhibited subclass hierarchies. Further, BSBM graphs and the Slegger ontology feature multiple inheritance. BSBM graphs satisfy (†). On Slegger we were unable to check this, as the data is not publicly shared; our understanding of the application though as described implies that (†) holds.

An older study [31] of many concrete RDF Schemas notes a high frequence of class hierarchies, of depth going up to 12, as well as a relatively high incidence of multiple inheritance; graphs with such schema benefit from $\mathtt{G}_{/\mathtt{WTH}}$ summarization when our hypothesis (†) holds.

4.3 Distributed algorithms

Below, we describe the algorithms we devised for building in a parallel fashion strong summaries (Section 4.3.1) and weak summaries (Section 4.3.2), as well as, typed strong and typed weak (Section 4.3.3). Their generic design makes them suitable for MapReduce-like frameworks. In Section 4.3.4 we discuss the adjustment of the algorithms wrt. Apache Spark framework as it was our implementation choice, and we used it to check the performance of the algorithms (5.2).

We assume the graph holds $|G|$ triples and we have M machines at our disposal.

Furthermore, all the algorithms perform two preprocessing steps. First consists of creating the set of class and property nodes. They need to be copied to the output as they are so their preservation is crucial. Second is an optimization step. The triples are integer-encoded in the same way as previously described in the centralized approach.

[3] http://dblp.uni-trier.de/

[4] http://slegger.gitlab.io/

4.3.1 Parallel computation of the strong summary

We compute the strong summary through a sequence of parallel processing jobs as follows.

1. We distribute *all (data and type)* triples of input graph equally among all the machines, e.g. using round robin approach, so that each m_i, $1 \leq i \leq M$ holds at most $\lceil \frac{|G|}{M} \rceil$ triples.

2. In Map job, each machine m_i for a given data triple t = s p o **emits** two pairs: $(\mathtt{s}, (\mathtt{source}, \mathtt{p}, \mathtt{o}))$ and $(\mathtt{o}, (\mathtt{target}, \mathtt{p}, \mathtt{s}))$, where source and target are two constant tokens (labels). Let us stress that *the data and type triples initially distributed to each machine $m_1, \ldots, m_M$ are kept (persisted) on that machine throughout the computation*. All other partial results produced are discarded after they are processed, unless otherwise specified.

3. In the corresponding Reduce job, for each resource $r \in \mathtt{G}$, all the data triples whose subject or object is r arrive on a same machine m_i. For each such r, m_i can infer some relationships (same source clique, same target clique) that hold between the data properties of G from incoming and outgoing edges of r. Formally, a *property relation information (or PRI, in short)* can take one of the following forms:

   Definition 15. (PROPERTY RELATION INFORMATION) *Let p_1, p_2 be two data properties in G. A PRI involving p_1, p_2 states that these properties are either (i) source-related (Definition 1), or (ii) target-related (Definition 1).*

   If m_i hosts, say, two data triples of the form $r\ p_1\ o_1$ and $r\ p_2\ o_2$, m_i can safely conclude that p_1 and p_2 are source-related (and similarly for target-related properties). The PRIs resulting from all the data triples hosted on m_i are gathered and de-duplicated.

   In the above, p_1 is not necessarily distinct from p_2. The interest of producing a PRI even for a property p_1 with itself is to provide the necessary information so that all the cliques of G are correctly computed in the steps below (even those consisting of a single data property p_1).

4. Each machine **broadcasts** its PRIs to all other machines *while also keeping its own PRIs*.

5. Based on this broadcast, each machine has the necessary information to compute the source and target cliques of G *locally*, and actually computes them[5].

   At the end of this stage, the cliques are known *and will persist on each machine until the end of the algorithm*, but we still need to compute: (i) all the (source clique, target clique) pairs which actually occur in G nodes, and (ii) the representation function and (iii) the summary edges.

6. The representation function can now be locally computed on each machine as follows:

   - For a given pair of source and target cliques (SC, TC), let N^{TC}_{SC} be an URI uniquely determined by SC and TC, such that a different URI is assigned to each distinct clique pairs: N^{TC}_{SC} will be the URI of the $\mathtt{G}_{/\mathtt{S}}$ node corresponding these source and target cliques.
   - For each resource r stored on m_i, the machine identifies the source clique SC_r and target clique TC_r of r, and creates (or retrieves, if already created) the URI $N^{TC_r}_{SC_r}$ of the node representing r in $\mathtt{G}_{/\mathtt{S}}$.

7. Finally, we need to build the edges of $\mathtt{G}_{/\mathtt{S}}$.

   (a) To summarize *data triples*, for each resource r whose representative N_r is known by m_i, and each triple (hosted on m_i) of the form $r\ p\ o$, m_i emits $(o, (\mathtt{p}, N_r))$. This triple arrives on the machine m_j which hosts o and thus already knows N_o. The machine outputs the $\mathtt{G}_{/\mathtt{S}}$ triple $N_r\ \mathtt{p}\ N_o$.

[5] In practice: (i) this can be implemented e.g., using Union-Find; (ii) this is redundant as only one of them could have done it and broadcast the result.

(b) To summarize *type triples*, for each resource r represented by N_r such that the type triple r *type* c is on m_i, the machine outputs the summary triple N_r *type* c[6].

The above process may generate the same summary triple more than once (and at most M times). Thus, a final duplicate-elimination step may be needed.

Algorithm correctness. The following observations ensure the correctness of the above algorithm's stages.

- Steps 1 to 4 ensure that each machine has the complete information concerning data properties being source- or target-related. Thus, each machine correctly computes the source and target cliques.
- Step 2 ensures that each machine can correctly identify the source and target clique of the resources r which end up on that machine.
- The split of the triples in Step 1 and the broadcast of source and target clique ensure that the last steps (computation of representation function and of the summary triples) yield the expected results.

4.3.2 Parallel computation of the weak summary

The algorithm for weak summarization bears many similarities with the one for the strong summary. Below, we only describe the steps that are different; in a nutshell, this algorithm exploits (is based upon) the observation that in the weak summary, each data property occurs only once.

4. In this step, instead of PRIs, the machines emit *Unification Decisions*:

   Definition 16. (UNIFICATION DECISION) *Given two data properties p_1, p_2, a unification decision is of one of the following forms: (i) p_1, p_2 have the same source node in $\mathtt{G}_{/\mathtt{W}}$; (ii) p_1, p_2 have the same target node in $\mathtt{G}_{/\mathtt{W}}$; (iii) the source of p_1 is the same as the target of p_2.*

   For instance, two triples of the form $r_1\ p_1\ r_2, r_2\ p_2\ r_3$ lead to the UD "the target of p_1 is the same as the source of p_2"; similarly, from $r_4\ p_1\ r_5, r_4\ p_2\ r_6$ lead to the UD "the source of p_1 is the same as the source of p_2" etc. In the above, just like for the PRIs, p_1 and p_2 can be the same or they can be different.

5. Each machine broadcasts its unique set of UDs while also keeping its own. Observe that for a group of k properties having, for instance, the same source, $k(k-1)/2$ UDs can be produced, however it suffices to broadcast $k-1$ among them (the others will be inferred by transitivity in the next step).

6. Each machine has the necessary information to compute the nodes and edges of $\mathtt{G}_{/\mathtt{W}}$ as follows:

   - Assume that for two sets IP, OP of incoming, respectively, outgoing data properties, we are able to compute a unique URI W_{IP}^{OP}, which is different for each distinct pair of property sets.
   - Build $\mathtt{G}_{/\mathtt{W}}$ with an edge for each distinct data property p in G; the source of this edge for property is $W_{\varnothing}^{\{p\}}$, while its target is $W_{\{p\}}^{\varnothing}$. All edges are initially disconnected, that is, there are initially $2 \times P$ nodes in $\mathtt{G}_{/\mathtt{W}}$.

[6]There is no need to flip the triple and send it to another map job because the object of type triple is already known to be a class node thus represented by itself.

- Apply each UD on the graph thus obtained, gradually fusing the nodes which are the source(s) and target(s) of the various data properties. This entails replacing each W node with one reflecting all its incoming and outgoing data properties known so far.

At the end of this process, each node has $\mathtt{G}_{/\mathtt{W}}$. We still need to compute the representation function.

7. On each machine holding a triple $r_1\ p\ r_2$, we identify the W nodes W_p, W^p in $\mathtt{G}_{/\mathtt{W}}$ which contain p in their outgoing, respectively, incoming property set. We output the $\mathtt{G}_{/\mathtt{W}}$ triple $W_p\ p\ W^p$.

8. The type summary triples are built exactly as in step 7b in the strong summarization algorithm.

4.3.3 Parallel computation of the typed strong and typed weak summaries

In this section we discuss necessary changes that need to be applied to the abovementioned algorithms that enable the computation of the typed counterparts of weak and strong summaries.

In both algorithms we need to modify some steps to reflect different type triples treatment. In particular we introduce new constant token `type` to be send in the step 2. We emit pairs corresponding to type triples only in the forward direction, e.g. we will send $(\mathtt{s}, (\mathtt{type}, \mathtt{p}, \mathtt{o}))$ but not $(\mathtt{o}, (\mathtt{type}, \mathtt{p}, \mathtt{s}))$. We do not emit pairs with tokens `source` nor `target` for type triples as they no longer contribute to property cliques discovery. Those type triples are then cached at the machines and not used until the step 6 in the strong summary algorithm or 6 in the weak algorithm respectively.

Afterwards, we want to compute the representation function locally. At each machine that received some type triples we group them together by the subject and create a unique id based on all the types. We notice here that we need to perform a step similar to 7a. It follows from the fact that locally we have complete information about the types of a given resource node r but we can't say without exchanging information with other machines whether the object of a triple with `source` token at that machine is a graph node that contributed to property cliques discovery or is it a typed node[7]

4.3.4 Apache Spark implementation specifics

System architecture

Spark is a big data processing engine that features analytics and extends traditional MapReduce framework. Spark supports various data sources that it can read from. Among them we can find Hadoop Distributed File System. Spark is able to read files stored there into memory and operate on them. In order to improve the performance and utilize all the resources it uses underlying scheduler to manage its jobs. In principle Spark, as a distributed system, can work with any scheduler, it even comes with a default one. However one of the common choices, that is the most compatible with HDFS, is Hadoop YARN scheduler (all three projects are developed by Apache software foundation). In our project we use Spark 2.3.0, YARN 2.9.0, Java OpenJDK 1.8.0_181 and Scala 2.11.

Spark's architecture consists of a partition of cluster's machines into worker nodes, which are responsible for executing the piece of a distributed computation, and a single driver node[8], which coordinates the run of an application and communicates with cluster manager (e.g. YARN scheduler). (Worker nodes also communicate through cluster manager). Spark uses a concept of executor, which is a separate process within a worker node with its own piece of data, assigned a fixed amount of available physical resources[9], that performs given tasks. Each Spark application consists of jobs that are divided into stages. Each stage is then further divided into tasks. As a scheduler, YARN uses containers, which almost directly translate to Spark executors. YARN tries to use the information about data locality while choosing the place for resource allocation. It can boost loading the files from HDFS as well as transition

[7] We know if it's a class or property node as we store such information in all 4 summary algorithms as mentioned in the introduction to the distributed algorithms

[8] In fact Spark uses the term driver program and this program can be launched as a process inside of the cluster or outside of the cluster depending on the deploy mode.

[9] Technically, Spark framework supports dynamic resource allocation but we decided not to use this feature.

between jobs (even though typically results of a map or reduce task aren't written back to HDFS unlike in MapReduce). Besides that, orthogonally, Spark collections e.g. RDDs are partitioned into logical chunks of data (partitions).

Spark Scala API

Spark comes with an API in 4 programming languages: Scala, Java, Python and R. For the convenience we chose Scala to implement our algorithms. This language provides Java-like type safety and many more features, some of which are coming from functional languages like matching or elegant case classes. On top of that, it is much more concise than Java so that the code is nearly as compact as equivalent Python implementation. Core Spark code is mainly written in Scala as well, which further supports our choice.

General programming paradigm of Spark

Spark uses a concept of Resilient Distributed Datasets (RDDs), which are fault-tolerant and immutable collections of data distributed among the cluster nodes. They can be operated on in parallel using two types of operations: transformations and actions. Transformations take an RDD as an input and produce a new RDD. Mainly they are equivalents of typical functional operations, examples of which can be: map, flatMap, filter, groupBy or reduceByKey. Actions return the value to the driver. Among them we can find operations like cache, collect, broadcast or saveToTextFile.

By default transformations are computed lazily. It means that they are queued along with the RDDs and only actions that follow transformation trigger they actual computation. The reason behind this design choice is that it lets Spark avoid some unnecessary computations and leave the room for optimizer.

Due to the lazy model of computations, Spark introduces execution graph which captures the operations order. It also doesn't cache any intermediate results by default. Every time some new RDD needs to be computed it applies all the operations on the path leading to the result. However a user may explicitly ask to cache/persist a given RDD. In this case Spark will store a copy of this RDD in the memory (at specified storage level), and in case of new computation that uses this RDD, it can read it from cache instead avoiding recompution. It is useful when some RDDs share the same path in the execution graph. We make use of caching feature in our implementation. Apart from that, Spark has also another construct called checkpoints. Here it is important to understand that checkpointing is aiming at preserving the state of the computations at some point in time in order to speed up computation recovery in the case of node failure (node crash). Its purpose then is clearly different.

As we have seen so far Spark supports map and reduce operations. It also comes with its own concept of broadcast variables. These are the collections of the data that are first gathered at the driver and then are broadcast (copies shipped over network) to all the cluster nodes. These collections are immutable and can only be used for local lookups.

In general data distributed by Spark and its underlying systems among the cluster nodes shouldn't be managed by the user. It means that the user operates on collections as black boxes (almost as if they weren't distributed) using transformations and actions providing a specification of the pipeline that leads to the resulting collection.

Implementation design choices

In Spark 2.0 DataFrames and Datasets APIs were unified. They are built on top of the RDDs and are meant to facilitate SQL-like processing by organizing data into columns (like in the relational model). In our case we don't benefit much from those extra features coming with SQL-like API. Taking into account the discussion in [16], we decided to stick to RDDs.

We need to state the differences between the generic algorithm and our Spark implementation. We construct our computation flow using RDDs as building blocks. Each step of the algorithms is mapped

into corresponding RDD. In this section we enumerate the most significant collections that has been defined in our code.

First of all, we load the input graph into `graph` RDD. Then, we preprocess it in order to create `dictionary`, `reverseDictionary`, `nonDataNodesBlacklist` and `encodedTriples` RDDs. `dictionary` stores a mapping between input graph nodes and their integer encoding, `reverseDictionary` is its reverse mapping and `encodedTriples` are the input graph triples integer-encoded. We collect class and property nodes in `nonDataNodesBlacklist` as this collection is broadcast and used for lookup.

Secondly, we create an RDD called `nodesGrouped` that represents input graph nodes together with their incoming and outgoing edges. Next, in the weak algorithm we create `unificationDecisions` RDD that is a collection of unification decisions, in the strong algorithm we create respectively `propertyRelationInformation` RDD. For those two RDDs we exclude (pre-filter) class and property nodes, and in case of typed summaries we exclude typed nodes too.

In both cases for `unificationDecisions` and `propertyRelationInformation` RDDs we gather those collections to the driver node. There we perform the necessary union-find step and having identified source and target cliques we broadcast the mappings to all the nodes in the cluster.

In the end of the processing we need to build the summary edges. In case of the weak summary we directly output summary edges as in step 6. For the other summaries we define another RDD called `representationFunction` that stores a mapping between the input graph node and its summary representant so that we modify step 7a. To actually compute the summary edges in this case we need to join `encodedTriples` on subject and separately on object with `representationFunction`.

Finally, in all the algorithms we need to decode the properties (edge labels). We do it by joining the RDDs resulting from nodes summarization in the previous step on property with `reverseDictionary`.

Actually we have a bottleneck in our algorithm. Whenever we compute unification decisions or property relation information we need to collect all the data to the driver node, where we further process them locally computing cliques and final mappings used for representing the nodes that are broadcast. We didn't find any way to distribute this computation and generic algorithm suffers from this step as well.

An important Spark-related issue that we encountered during the implementation of the integer-encoding preprocessing step was that resulting summary was incorrect unless we cached `dictionary` RDD. The problem comes from the transformation called zipWithUniqueId used to create `dictionary` RDD. Without caching `dictionary`, when we computed `reverseDictionary` this transformation was computed from scratch. Since there is no other guarantee then uniqueness of the ids being appended to the values, each time we were ending up with `reverseDictionary` not being a proper reverse mapping for `dictionary` RDD (String labels of subjects, properties and objects were given different ids in different runs). Therefore caching for `dictionary` RDD is here indispensable.

We use caching in other parts of our code, whenever there are some common pathis in the execution graph, to optimize the whole application execution.

Chapter 5

Experiments

This chapter describes the experiments conducted in order to validate the performance of the algorithms for RDF graphs summaries. Section 5.1 presents the empirical study of the centralized algorithms. Section 5.2 describes the distributed algorithms trade-offs and the configuration parameters' choice, and provides the comparison between the computation time in the single-machine and cluster settings.

5.1 Centralized algorithms experiments

Algorithms and settings. We have implemented our algorithms as well as `1fb` summarization in Java 1.8. We also report below experiments conducted prior to my arrival in the team, based on pre-existing code implementing the `fw`, `bw`, `fb` summaries. We used the recent [29] algorithm for `fw`, `bw` and `fb` and devised our simple algorithm for `1fb`. We deployed them using PostgreSQL v9.6 to store RDF triples in an integer-encoded triple table, indexed by `s`, `p` and `o`; the server had 30 GB of shared buffers and 640 MB working memory. The JVM had 90 GB of RAM. We used a Linux server with an Intel Xeon CPU E5-2640 v4 @2.40GHz and 124 GB RAM.

Real datasets	$\|G\|$	$\|G^\infty\|$	$\|S_G\|$	$\#p$	$\#C$
DBLP	88,153,334	88,153,334	0	26	14
DBpedia Person	7,889,268	7,889,268	0	9	1
INSEE Geo	369,542	1,112,803	196	53	113
Springer LOD	145,136	213,017	40	26	11
Nobel Prizes	80,757	109,901	35	45	26
Synth. datasets	$\|G\|$	$\|G^\infty\|$	$\|S_G\|$	$\#p$	$\#C$
LUBM [20] 1M	1,227,744	1,227,744	0	17	21
LUBM 10M	11,990,059	11,990,059	0	17	21
LUBM 100M	114,355,295	114,355,295	0	17	21
BSBM [2] 1M	1,000,708	1,009,138	150	38	159
BSBM 10M	10,538,484	10,628,484	584	38	593
BSBM 100M	104,115,556	105,315,556	2,010	38	2019
BSBM 138M	138,742,476	140,342,476	2,412	38	2,421

Table 5.1: Datasets used in experiments.

Datasets. We have experimented with numerous real and synthetic datasets, from 10^5 to $1.4 \cdot 10^8$ triples; the largest file is 36.5 GB on disk. Table 5.1 shows for each graph its number of triples $|G|$, the number of triples in the saturated graph $|G^\infty|$, the number of schema triples $|S_G|$, and the number of distinct data properties $\#p$ and classes $\#C$.

Compression factors. Figure 5.1 shows the ratio $|G|/|G_{/\equiv}|$, called the *compression factor* $cf_\equiv$ for our graphs. To our summaries we added `1fb`, `fw`, `bw` and `fb`; we plot `fw` and `fb`, as `bw` was somewhere in-

between. Some fw and bw summarizations ran out of memory or were stopped after more than 2 hours. For W and S, cf is close to or above 10^3, and reaches $3 \cdot 10^6$ in some cases; in all our experiments, $cf_{\mathtt{W}}$ was the highest. In contrast, $cf_{\mathtt{fb}}$ rounds to 1 up to 3, since full bisimilarity is rare. Since $\equiv_{\mathtt{fw}}$ is a weaker condition and $\equiv_{\mathtt{1fb}}$ even weaker, $cf_{\mathtt{fw}}$ and especially $cf_{\mathtt{1fb}}$ are higher, but still up to 5 times lower than $cf_{\mathtt{W}}$. *In all our experiments, the weak summary is the most compact;* S, TW *and* TS *are close, followed by* 1fb, *from* 2.9 *to* 6.9 *times larger*. Conversely, full bisimulation achieves little to no compression, while fw (which has the drawback of being asymetric) compresses less than 1fb.

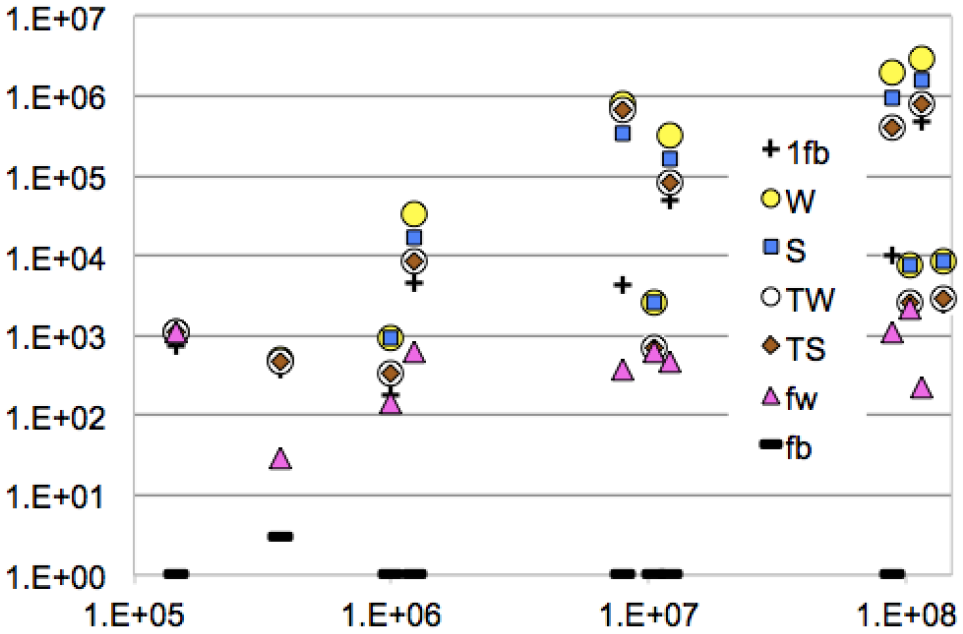


Figure 5.1: Summarization compression factors for various graph sizes |G|.

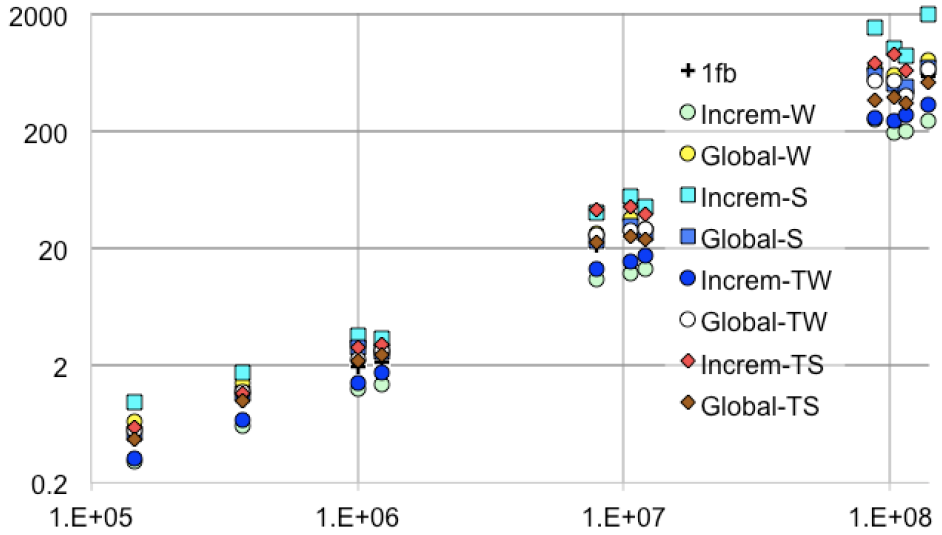


Figure 5.2: Summarization time (s) for various graph sizes |G|.

Summarization time. The time to build $\mathtt{G}_{/\mathtt{W}}$, $\mathtt{G}_{/\mathtt{S}}$, $\mathtt{G}_{/\mathtt{TW}}$ and $\mathtt{G}_{/\mathtt{TS}}$ using the global and the incremental algorithms, as well as the time to build $\mathtt{G}_{/\mathtt{1fb}}$, are plotted as a function of |G| in Figure 5.2; both axes are in log scale. For each summary type, the summarization time is roughly linear in the size of G,

confirming the expectations stated in Section 4.1; they range from 200 ms to 34.5 minutes (increm-S on the BSBM138M). Increm-W is the fastest overall; it traverses G only once, and it has relatively little processing to do, thus it is faster than global-W which performs several passes, but does not need to replace node representatives. 1fb summarization time is close, then we see the global S, TW and TS, and finally incremental S which, as we explained, is quite complex. The fact that increm-W is the cheapest and increm-S the most expensive show that the former may be I/O-bound, while the latter (with the same I/O cost) is CPU (and memory)-bound. Since increm-S is rather expensive per-triple, it is more efficient to first summarize a graph using global-S, and call increm-S only to maintain it later. This is significantly faster: for instance, global-S on BSBM138M takes only 11.85 minutes. Increm-TS is often faster than increm-S because typed nodes do not lead to splits during TS summarization.

Shortcut speed-up. Table 5.2 shows the time to build $(\mathtt{G}^{\infty})_{/\equiv}$ in two ways: (i) *direct*, i.e., saturate G then summarize, denoted $dt_{\equiv}$, and (ii) *shortcut*, summarize G, then saturate the summary and summarize again, denoted $st_{\equiv}$. We define the **shortcut speed-up** $x_{\equiv}$ for $\equiv \in \{\mathtt{W}, \mathtt{S}\}$ as $(dt_{\equiv} - st_{\equiv})/dt_{\equiv}$ and report it also in the table. The speed-up is highest for $\mathtt{G}_{/\mathtt{W}}$ (up to almost 98%) and $\mathtt{G}_{/\mathtt{S}}$ (up to 95%): this is a direct consequence of their good compression. Indeed, $dt_{\equiv}$ includes the time to summarize $\mathtt{G}^{\infty}$, while $st_{\equiv}$ includes the time to summarize $(\mathtt{G}_{/\equiv})^{\infty}$; the smaller this is, the higher $x_{\equiv}$. The table confirms the practical interest of the shortcut (Section 2.5) for summarizing the full semantics of a graph G.

Dataset	$dt_{\mathtt{W}}$ (s)	$st_{\mathtt{W}}$ (s)	$x_{\mathtt{W}}$ (%)	$dt_{\mathtt{S}}$ (s)	$st_{\mathtt{S}}$ (s)	$x_{\mathtt{S}}$ (%)
INSEE Geo	35.85	0.81	97.73	38.59	1.95	94.96
Springer	3.96	0.45	88.60	4.59	1.159	74.73
Nobel	2.13	0.41	80.56	2.60	0.75	71.09
BSBM1M	5.45	1.48	72.85	9.35	3.82	59.20
BSBM10M	71.63	12.67	82.32	142.46	55.64	60.94
BSBM100M	989.00	198.00	79.98	1715.40	1030.36	39.93
BSBM138M	1393.69	251.93	81.92	3627.19	2049.22	43.50

Table 5.2: Shortcut experiments.

Experiment conclusion. Our experiments have shown that our four summaries can be built efficiently, and they reduce graph sizes by factors hundreds up to $3 \cdot 10^6$; they are *two to three orders of magnitude more compact than summaries based on full bisimulation*; $\mathtt{G}_{/\mathtt{W}}$ is the most compact, and the other three summaries we introduced are close (within a factor of 3). Finally, among the summaries which admit a shortcut ($\mathtt{G}_{/\mathtt{W}}$, $\mathtt{G}_{/\mathtt{S}}$, $\mathtt{G}_{/\mathtt{fb}}$ and $\mathtt{G}_{/\mathtt{fw}}$), *the shortcut speed-up is up to 98% for* $\mathtt{G}_{/\mathtt{W}}$, *and 95% by* $\mathtt{G}_{/\mathtt{S}}$. All our algorithms scale linearly with $|\mathtt{G}|$; increm-W is the fastest, while increm-S is the slowest. Overall, if summary size or building time are a critical concern, we recommend using $\mathtt{G}_{/\mathtt{W}}$; otherwise, $\mathtt{G}_{/\mathtt{S}}$ gives finer-granularity information. TW and TS summaries should be used when data types are deemed most important in order to understand the data. However, to summarize $\mathtt{G}^{\infty}$, only the direct path (saturate, then summarize) is available for these.

5.2 Distributed algorithms experiments

5.2.1 Cluster setup

We are using a cluster of 6 machines, each of which is equipped with an Intel Xeon CPU E5-2640 v4 @2.40GHz and 124GB RAM. Each machine has 20 physical CPU cores, however we use the cluster with a hyper-threading option enabled. This option gives an operating system effectively 40 cores for resource allocation. All machines in this cluster are connected to a switch using 10 Gigabit Ethernet. We give to Spark and YARN 100GB of RAM and 36 cores at each machine. We leave some fraction of the memory (remaining 24GB) and 4 CPU cores for the operating system.

5.2.2 Configuration

We recall here that M denotes the number of machines, more precisely workers (we don't count the driver node). We introduce useful notation that helps us show the trade-offs in the configuration of Spark application. Variables are given a possible ranges of the values they can take along with the explanation.

- Number of cores per machine
  $C_M = 36$
  We pick all the available resources in order to maximize the performance.

- Number of cores per executor
  $C_E \in \{1, \ldots, 36\}$
  Upper bound when $C_E = C_M$. Usually we pick here $C_E = 4$ following general guidelines not to give more cores per executor.

- Amount of memory per machine
  $R_M = 100$GB
  Here as well, we pick all the available memory to maximize the performance.

- Amount of memory per executor
  $R_E \geq 2.78$GB
  Since there are C_M cores per machine, so there can be at most C_M executors per machine, so they will use at least $\frac{R_M}{C_M}$, in general $R_E = \frac{R_M \cdot C_E}{C_M}$. We set the lower bound for R_E as a minimal memory of the YARN container, within the memory limit for executor we need to hold out around 1GB for a memory overhead (memory for JVM in YARN).

- Number of executors per machine
  $E_M \in \{1, \ldots, 36\}$
  We can have at most C_M executors per machine. We also have $C_M = C_E \cdot E_M$ and $R_M = R_E \cdot E_M$. Usually we pick $E_M = 9$.

- Total number of executors
  $E \in \{1, \ldots, 180\}$
  Also $M \leq E$ since we assign at least 1 executor per machine and not more than E_M per machine, thus $E = E_M \cdot M$.

- Number of partitions
  $P \in \{1, \ldots, 180\}$
  $P = \alpha E$, where $\alpha \in \{2, 3, 4\}$. With the choice of the range of α parameter value we rely on another popular rule of thumb, usually choosing $\alpha = 4$ in our case.

5.2.3 Speed up thanks to the increase of the degree of parallelism

From now on, we fix the configuration of the Spark application to:
$M = 5$, $C_E = 4$, $E_M = 9$, $R_E = 11$GB, $P = 180$.

Figures 5.3 and 5.4 depict the computation times for distributed algorithms with respect to the number of executors and divided by summary type. In this place we study an impact of M and E_M parameters or in other words E parameter and we fix our analysis on the BSBM 10M dataset. The former figure shows the total time including loading, preprocessing, summarization and saving the output file, whereas the latter shows isolated summarization time. We can see that summarization time takes only a small fraction of the whole algorithm (pay attention to the time units with respective figures). Additionally, despite the bottleneck nature of the procedure, it clearly scales with the increase of the degree of parallelism, in this case being the number of executors.

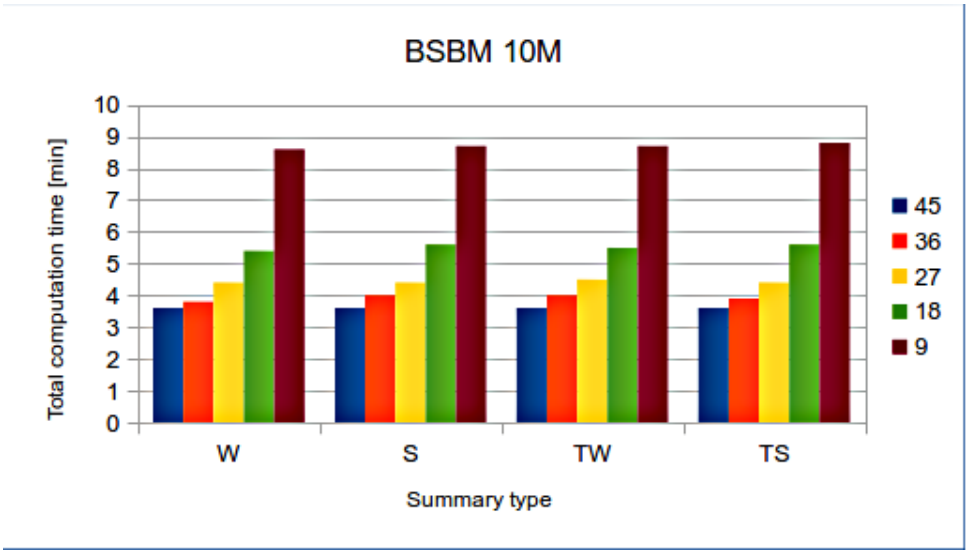


Figure 5.3: Total computation time for various number of executors.

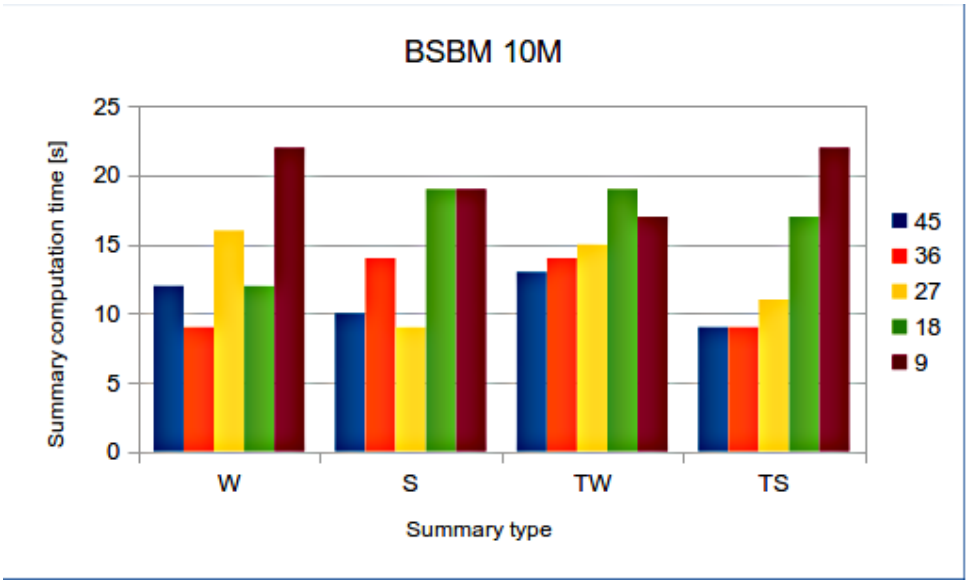


Figure 5.4: Summarization time for various number of executors.

Next analysis we perform with respect to the size of the dataset given discussed fixed configuration of the parameters. Figures 5.5 and 5.6 show the computation times for distributed algorithms with the increasing dataset size (from 1 million triples up to 100 million triples, all coming from BSBM benchmark) and divided by summary time. Similar to previous graphs, we capture the difference between total computation time and summarization time. They again exhibit similar ratios accordingly. What is more, we can see a stable growth with the increase of the size. Datasets differ in orders of magnitude, we therefore use log-scale on time axis.

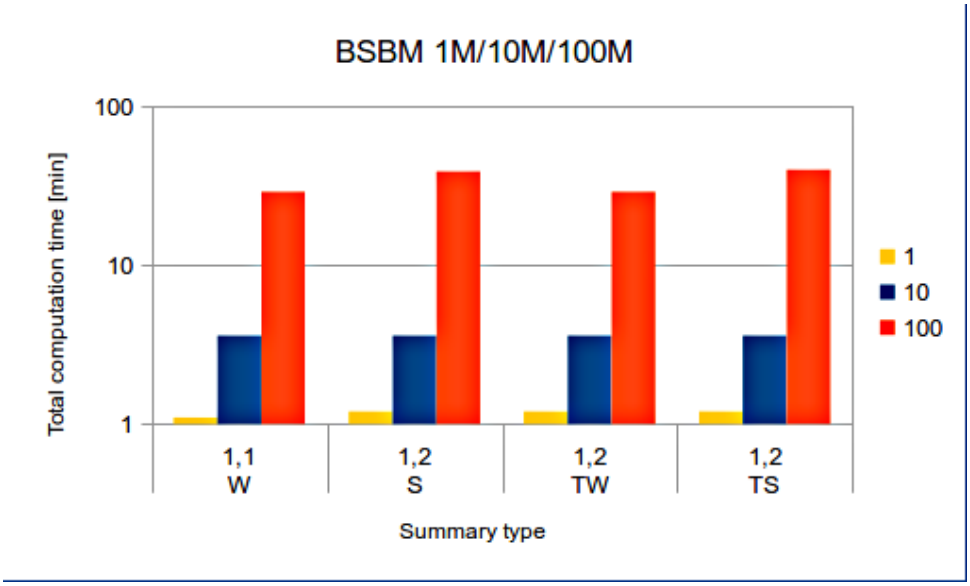


Figure 5.5: Total computation time for datasets of different size.

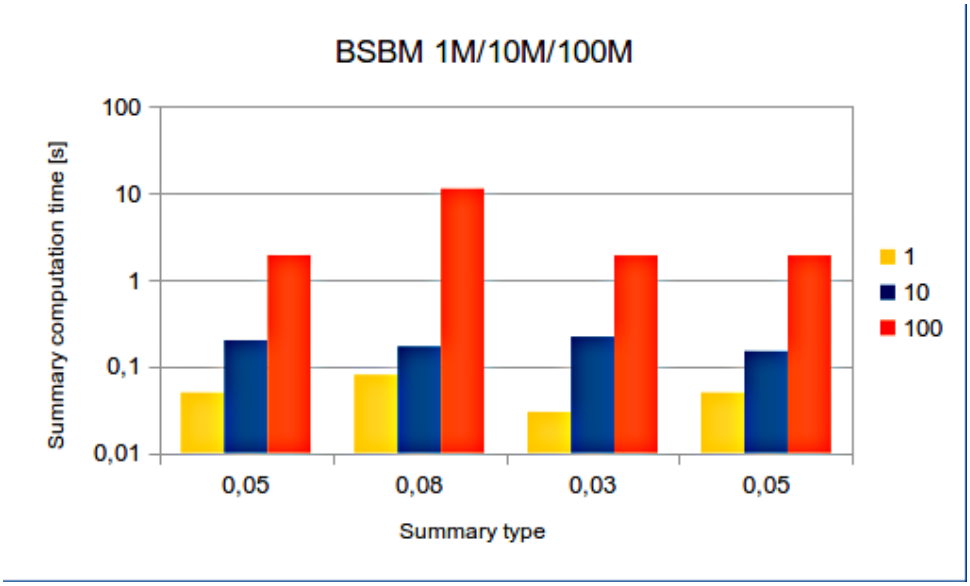


Figure 5.6: Summarization time for datasets of different size.

Chapter 6

Related Work

Verious graph summaries has been proposed in the past, as shown in the recent survey [30]. They may rely on *graph structure*, *graph values* or *graph statistics*. In particular exploring the number of distinct values for a given property or the number of edges adjacent to a node. The purpose of the summaries is to reflect the graph structure, possibly in the lossy manner. They may focus on the whole graph or a part of it. Summaries have been used to support *indexing and query processing*, that is, allow a query to be partially or fully evaluated on the smaller summary instead of $\mathtt{G}$. They can also be used as *a static analysis tool*, e.g., to detect empty-answer queries without actually consulting $\mathtt{G}$.

In this work, we study *structural quotient summaries*, which are *complete* and *representative* as discussed in Section 2.2. Quotient summaries most widely studied in the literature are based on *bisimulation* [32, 12, 26, 15]; they can be built in $O(|\mathtt{G}| \cdot \log(N))$, where N is the number of nodes in $\mathtt{G}$. Forward-and-backward bisimulation $\equiv_{\mathtt{fb}}$, symmetric w.r.t. edge directions, is most suited to RDF. However, it is well known, e.g., shown in [29], that heterogeneous graph nodes are very rarely $\equiv_{\mathtt{fb}}$, thus $\equiv_{\mathtt{fb}}$ summaries barely compress $\mathtt{G}$ (recall Figure 5.1). $\equiv_{\mathtt{1fb}}$ is more permissive, and can be seen as the closest competitor of $\equiv_{\mathtt{W}}$ and $\equiv_{\mathtt{S}}$; we have shown $\equiv_{\mathtt{1fb}}$ still leads to summaries several times larger than ours.

We view our property clique-based summaries as **complementary to** bisimulation-based ones: ours cope better with the heterogeneity of RDF graphs, thus are more suited for visualisation (they were precisely chosen for that in LODAtlas [34]), while bisimulation-based ones lead to larger summaries but allow, e.g., finding "all nodes having properties a and b" as those represented by a $\mathtt{G}_{/\mathtt{1fb}}$ node having an a and a b. If one of our summaries is used for this task, a *superset* of the desired nodes is obtained. Bisimulation- and clique-based summaries each have distinct advantages, and can be used side-by-side for different purposes.

With respect to distributed way of computing the summaries, [13] and [29] study optimizations to parallel bisimulation algorithm using multi-core graph processing framework and enhance SPARQL engines by enabling summary-based exploration and navigational query optimization. [3] shows approximate graph summarization algorithms that can be efficiently computed e.g., in MapReduce framework.

Other types of summaries, such as Dataguides [19] are not quotients, as a graph node may be represented by *more than one* node. A Dataguide may be larger than the original graph, and its construction has worst-case exponential time complexity in the size of $\mathtt{G}$.

With a focus farther from our work, [11] introduces an *aggregation* framework for OLAP on labeled graphs, while we focus on representing complete graph structure and semantics. [10] builds a set of randomized summaries to be *mined* instead of the original graph for better performance, with guaranteed bounds on the information loss. Focusing on RDF graphs, [36, 38, 13] study bisimulation-based RDF quotient summaries, providing efficient parallel summarization algorithms [13] and showing they are representative [36]. However, these summaries ignore RDF saturation, and thus its interaction with summarization. Summaries based on clustering [21], user-defined aggregation rules [35], mining [10], and identification of frequent subtrees [40] are not complete and/or require user input. [33] introduces a *simulation* RDF quotient based on *triple (not node) equivalence*. [3] studies simple methods for summarizing $\mathtt{D}_{\mathtt{G}}$, i.e. the data triples only.

We had demonstrated [6] and (informally) presented $\mathtt{G}_{/\mathtt{W}}$ and $\mathtt{G}_{/\mathtt{TW}}$ in a short "work in progress" paper [5], with procedural definitions (not as quotients). The shortcut was briefly described in a poster [8]. In [22], we propose a type-first summarization technique which exploits subclass hierarchies; beyond the quotient summary framework which it shares, it is orthogonal to the present work.

Chapter 7

Conclusion

This internship work focused on the quotient summaries of RDF graphs. Such synopses, being RDF graph themselves too, are concise and informative. They are based on structural similarity which makes the trade-off between insightful representation and compact size possible. They come in different flavors, exploring the structure in weak and strong summaries, as well as additional schema information that may be present in the graph. We pay a special attention to type triples exploring data-then-type and type-then-data, with the latter being further extended in order to capture and utilize possible hierarchical relations in the schema.

We perform a thorough experimental study to show the compactness of the summaries as well as almost-linear computation time. Summaries are able to compress input graphs by up to 6 orders of magnitude. We provide centralized versions of all the summaries. Each of the centralized algorithms comes in two variants: global or incremental. We further provide distributed version of summarization algorithms, abstractly described in MapReduce framework language, and concretized in Apache Spark framework implementation. On top of that, we examine shortcut, an efficient procedure to summarize graphs in the presence of schema information. We show significant speed-up up to 98%.

Quotient RDF summaries are proved to help users and application discover the graph structure. They have been integrated into LODAtlas [34], a data exploration and visualization platform. Their compactness was a strong argument for choosing them. They also found their applications in summary-based keyword search in RDF graphs.

During the implementation of distributed algorithms, another interesting idea come to our minds but we haven't been able to validate its usefulness and we leave it as a potential future work. It is possible that instead of collecting all the pieces of the information about the input graph distributed among the cluster nodes, we could instead follow divide and conquer strategy. We would cut the graph into smaller subgraphs and summarize them recursively. Probably it would be possible for a weak summary and plausibly limited only to that one for the reasons similar to the splitting issues encountered in the incremental strong algorithm. It is most likely linked with associativity of a given summary.

Bibliography

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] C. Bizer and A. Schultz. The Berlin SPARQL Benchmark. *Int. J. Semantic Web Inf. Syst.*, 5(2):1–24, 2009.

[3] S. Campinas, R. Delbru, and G. Tummarello. Efficiency and precision trade-offs in graph summary algorithms. In *IDEAS*, 2013.

[4] Š. Čebirić, F. Goasdoué, P. Guzewicz, and I. Manolescu. Compact Summaries of Rich Heterogeneous Graphs. Research Report RR-8920, INRIA Saclay ; Université Rennes 1, June 2017.

[5] Š. Čebirić, F. Goasdoué, and I. Manolescu. Query-oriented summarization of RDF graphs. In *BICOD*, 2015.

[6] Š. Čebirić, F. Goasdoué, and I. Manolescu. Query-oriented summarization of RDF graphs (demonstration). *PVLDB*, 8(12), 2015.

[7] Š. Čebirić, F. Goasdoué, and I. Manolescu. Query-Oriented Summarization of RDF Graphs. In *BDA (Bases de Données Avancées)*, 2016.

[8] Š. Čebirić, F. Goasdoué, and I. Manolescu. A framework for efficient representative summarization of RDF graphs. In *ISWC (poster)*, 2017.

[9] Š. Čebirić, F. Goasdoué, and I. Manolescu. Query-Oriented Summarization of RDF Graphs. Research Report RR-8920, INRIA Saclay ; Université Rennes 1, June 2017.

[10] C. Chen, C. X. Lin, M. Fredrikson, M. Christodorescu, X. Yan, and J. Han. Mining graph patterns efficiently via randomized summaries. *PVLDB*, 2(1), 2009.

[11] C. Chen, X. Yan, F. Zhu, J. Han, and P. S. Yu. Graph OLAP: towards online analytical processing on graphs. In *ICDM*, 2008.

[12] Q. Chen, A. Lim, and K. W. Ong. $D(K)$-index: An adaptive structural summary for graph-structured data. In *SIGMOD*, 2003.

[13] M. P. Consens, V. Fionda, S. Khatchadourian, and G. Pirrò. S+EPPs: Construct and explore bisimulation summaries + optimize navigational queries; all on existing SPARQL systems (demonstration). *PVLDB*, 8(12), 2015.

[14] M. P. Consens, R. J. Miller, F. Rizzolo, and A. A. Vaisman. Exploring XML web collections with DescribeX. *TWEB*, 4(3), 2010.

[15] M. P. Consens, R. J. Miller, F. Rizzolo, and A. A. Vaisman. Exploring XML web collections with DescribeX. *ACM TWeb*, 4(3), 2010.

[16] J. Damji. A Tale of Three Apache Spark APIs: RDDs, DataFrames, and Datasets. https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html, 2016. [Online; accessed 31-August-2018].

[17] W. Fan, J. Li, X. Wang, and Y. Wu. Query preserving graph compression. In *SIGMOD*, 2012.

[18] F. Goasdoué, I. Manolescu, and A. Roatiş. Efficient query answering against dynamic RDF databases. In *EDBT*, 2013.

[19] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, 1997.

[20] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *J. Web Sem.*, 3(2-3), 2005.

[21] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald. Using graph summarization for join-ahead pruning in a distributed RDF engine. In *SWIM*, 2014.

[22] P. Guzewicz and I. Manolescu. Quotient RDF Summaries Based on Type Hierarchies. In *DESWeb (Data Engineering meets the Semantic Web) Workshop*, Paris, France, Apr. 2018.

[23] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing simulations on finite and infinite graphs. In *FOCS*, 1995.

[24] D. Hovland, R. Kontchakov, M. G. Skjæveland, A. Waaler, and M. Zakharyaschev. Ontology-based data access to slegge. In *ISWC*, 2017.

[25] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering indexes for branching path queries. In *SIGMOD*, 2002.

[26] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering indexes for branching path queries. In *SIGMOD*, 2002.

[27] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *ICDE*, 2002.

[28] S. Khatchadourian and M. P. Consens. ExpLOD: Summary-based exploration of interlinking and RDF usage in the linked open data cloud. In *ESWC*, 2010.

[29] S. Khatchadourian and M. P. Consens. Constructing bisimulation summaries on a multi-core graph processing framework. In *GRADES*, 2015.

[30] Y. Liu, T. Safavi, A. Dighe, and D. Koutra. Graph summarization methods and applications: A survey. *ACM Comput. Surv.*, 51(3):62:1–62:34, June 2018.

[31] A. Magkanaraki, S. Alexaki, V. Christophides, and D. Plexousakis. Benchmarking RDF schemas for the semantic web. In *ISWC*, 2002.

[32] T. Milo and D. Suciu. Index structures for path expressions. In *ICDT*, 1999.

[33] F. Picalausa, Y. Luo, G. H. L. Fletcher, J. Hidders, and S. Vansummeren. A structural approach to indexing triples. In *ESWC*, 2012.

[34] E. Pietriga, H. Gözükan, C. Appert, M. Destandau, Šejla Čebirić, F. Goasdoué, and I. Manolescu. Browsing linked data catalogs with LODAtlas. In *Int'l. Semantic Web Conference (ISWC), Resources track*, 2018.

[35] M. Rudolf, M. Paradies, C. Bornhövd, and W. Lehner. SynopSys: large graph analytics in the SAP HANA database through summarization. In *GRADES*, 2013.

[36] A. Schätzle, A. Neu, G. Lausen, and M. Przyjaciel-Zablocki. Large-scale bisimulation of RDF graphs. In *SWIM*, 2013.

[37] Y. Tian, R. A. Hankins, and J. M. Patel. Efficient aggregation for graph summarization. In *SIGMOD*. ACM, 2008.

[38] T. Tran, G. Ladwig, and S. Rudolph. Managing structured and semistructured RDF data using structure indexes. *IEEE TKDE*, 25(9), 2013.

[39] W3C. Resource description framework. http://www.w3.org/RDF/.

[40] P. Zhao, J. X. Yu, and P. S. Yu. Graph indexing: Tree + delta >= graph. In *VLDB*, 2007.